**Generics**

**📘 What is Generics in Java?**

**Generics** is a feature in **Java** that allows you to write **classes, interfaces, and methods** that can operate on **different data types** while maintaining **type safety**.

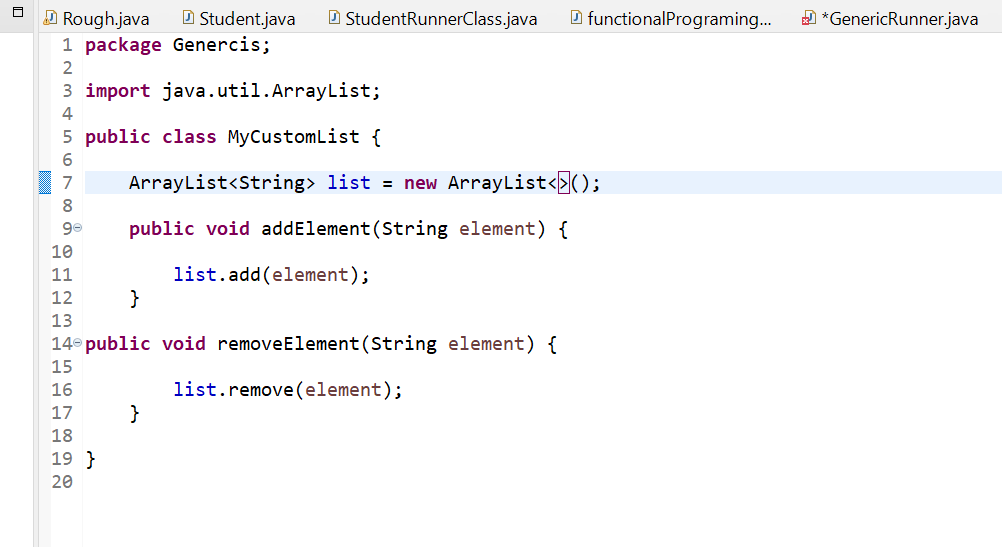
**✅ Why Do We Need Generics?**

Without generics, you would have to create **separate classes** for each data type, which leads to **code duplication** and **type-safety issues**.  
Generics provide a way to **write a single class or method** that works for **all data types**, improving **code reusability** and **maintaining type safety**.

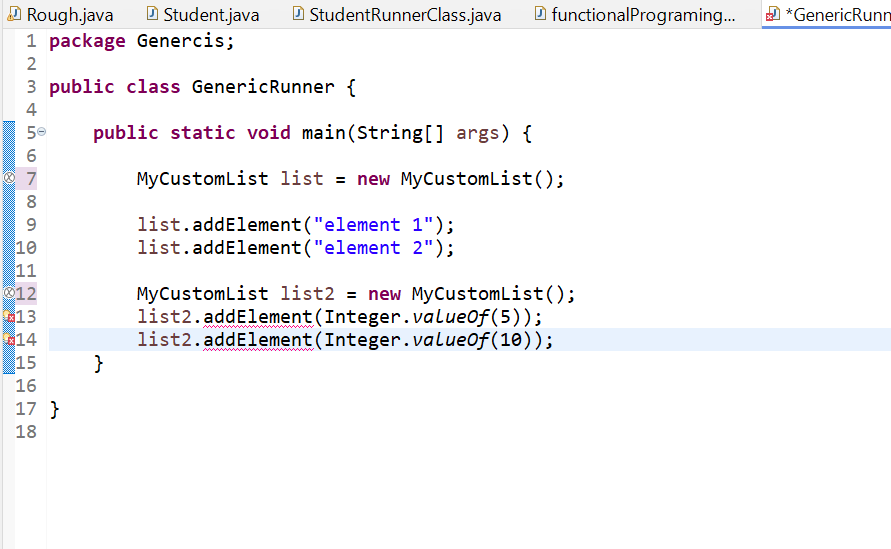
Let me explain this with a **real-world example** using your **MyCustomList** class.

**🖥️ Example Without Generics (Type-Specific Class)**

**MyCustomList Class Without Generics:**

****

GenericRunner Class Without Generics:



We can see the last two line has compilation Error

**🔎 What’s the Issue?**

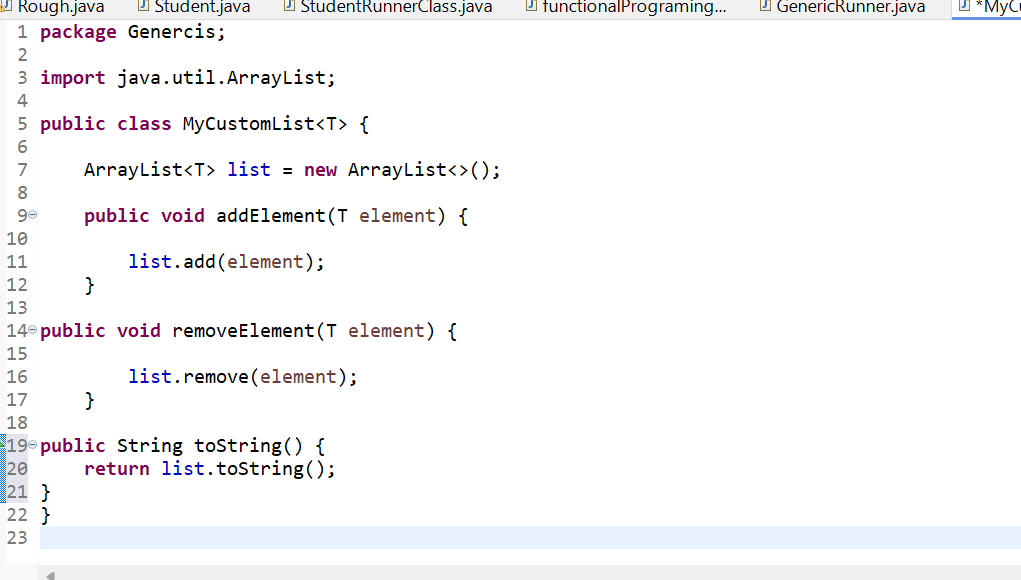
In the above code:

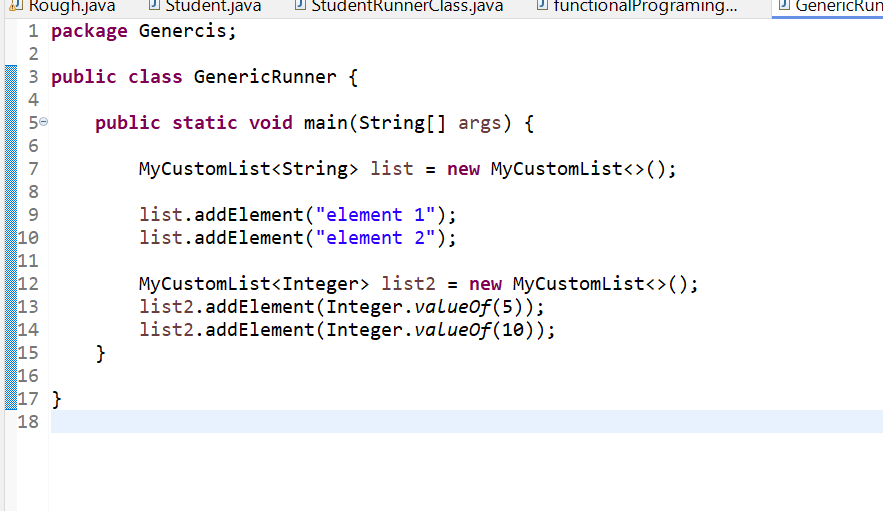
* The **MyCustomList** class is specifically designed to hold **String values**.
* When we try to add **Integer values** to **list2**, it throws an **error** because **Java enforces type safety**.

**✅ Solution: Use Generics**

We can solve this issue by making **MyCustomList** a **generic class**.  
This way, we can **specify the data type** when we create an object of the class.

🖥️ **Modified MyCustomList Class with Generics:**

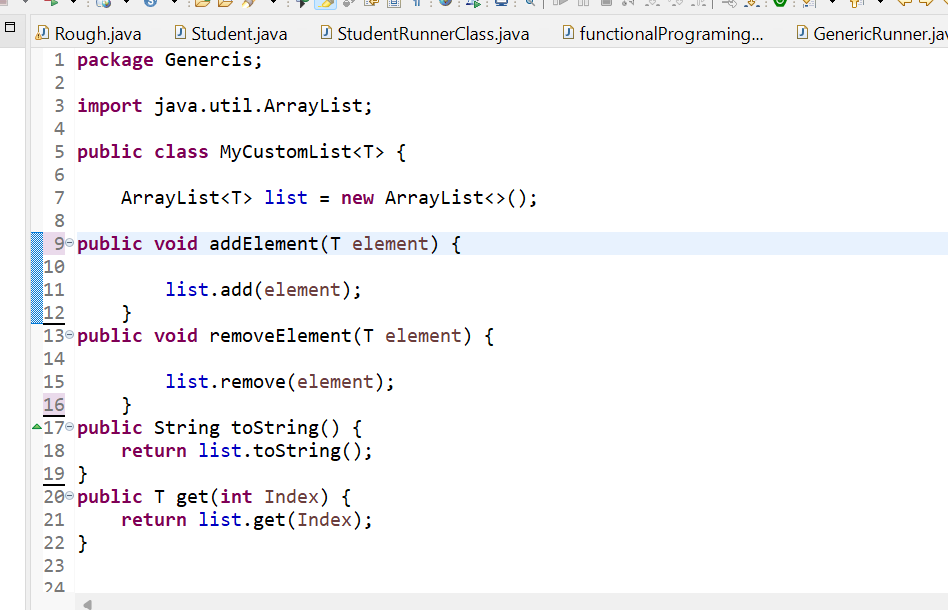


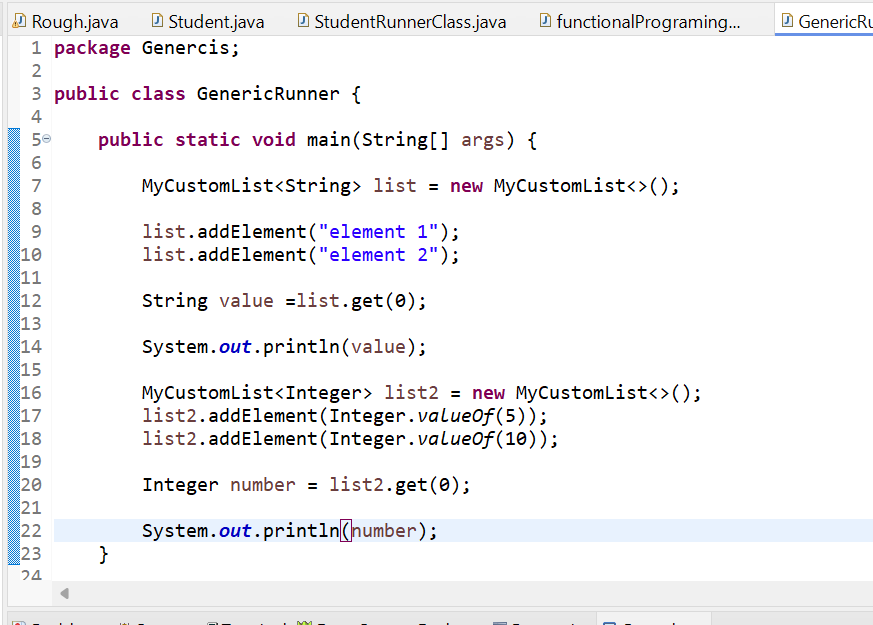


**✅ Benefits of Using Generics:**

1. **Type Safety**:  
   Generics ensure that you **only add elements of the specified type**, avoiding **ClassCastException** at runtime.
2. **Code Reusability**:  
   You can use the **same class or method** for different data types without duplicating code.
3. **Compile-Time Checking**:  
   Generics allow **type checking at compile-time**, reducing runtime errors.

**2.Custom Return With Generic List Method**

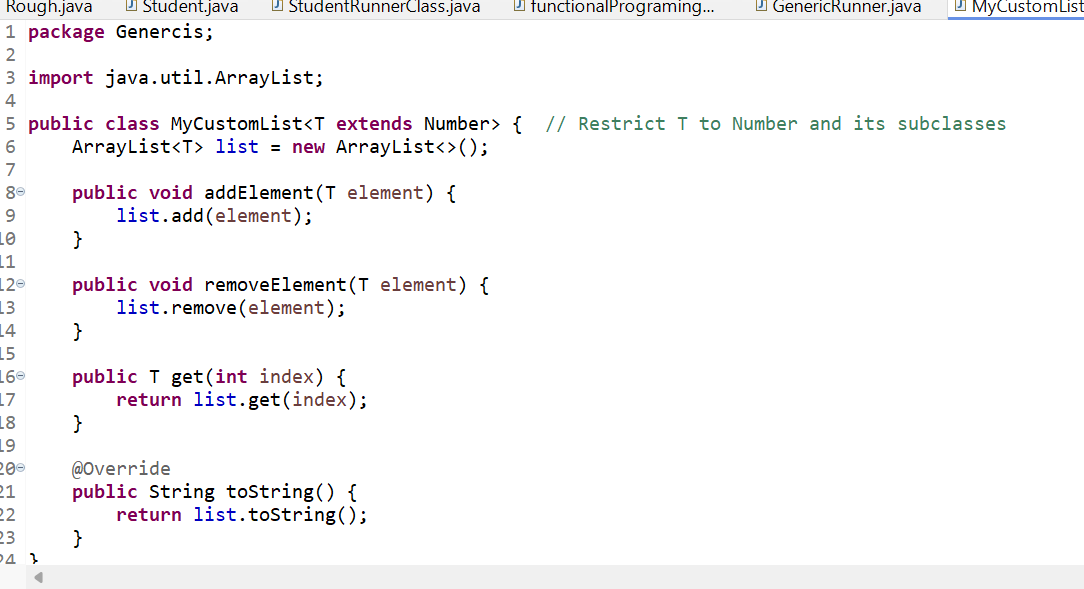


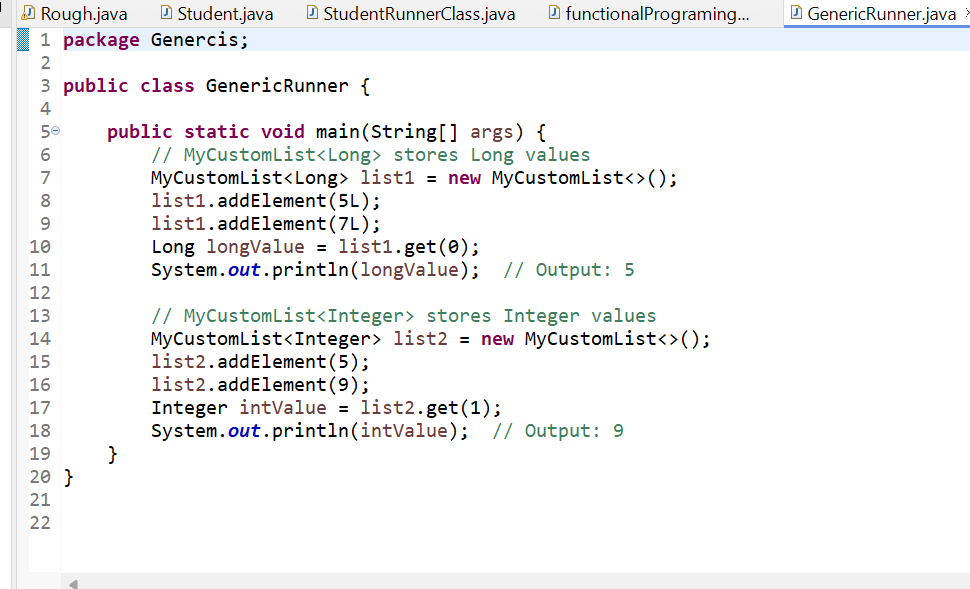


**✅ Implementing Type Restrictions on Generics**

By default, generics in Java allow any data type. But sometimes, you want to **restrict the types** that can be used. For example, you might want your custom list to only accept **numeric values** (like Integer, Double, Long, etc.).

This is where **type restrictions** come into play.





**✅ Explanation of Flow:**

1. **MyCustomList class is defined with a type restriction:**
   * T extends Number means that T can only be **Number or its subclasses** (Integer, Double, Float, Long, etc.).
2. **List1 and List2 are created:**
   * list1 stores **Long values**.
   * list2 stores **Integer values**.
3. **Using get() method to retrieve values:**
   * **list1.get(0)** returns 5L (Long).
   * **list2.get(1)** returns 9 (Integer).

**✅ Generic Methods**

You can also create **generic methods** to perform operations on **any type**.

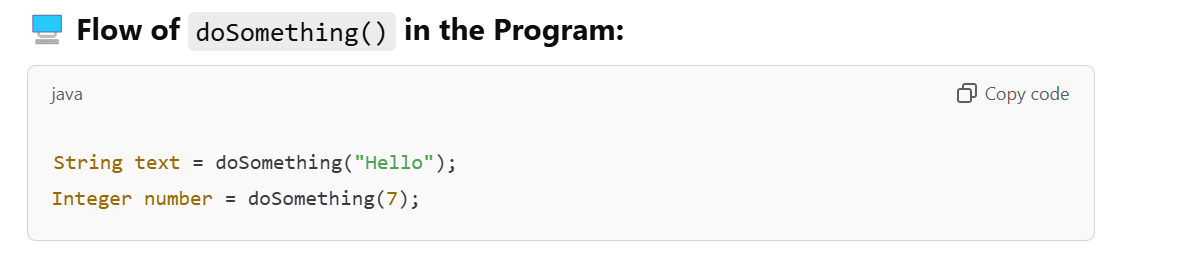
**✅ Program Flow Explanation**

**🔎 Method 1: doSomething()**



**💡 What It Does:**

* This is a **generic method** that accepts a parameter of **any type X** and **returns the same value**.
* The **<X>** before the method indicates that this method is **generic**.
* The type **X** is **inferred from the arguments** when the method is called.

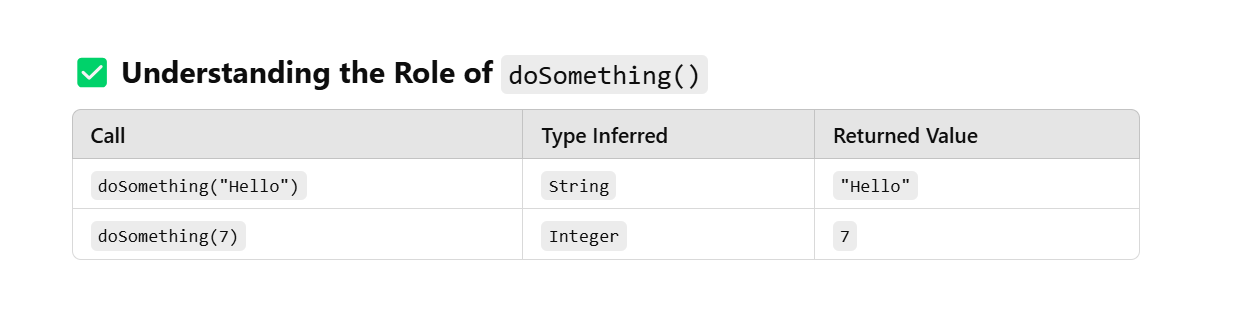


**🔧 Step 1: Call doSomething("Hello")**

* The method is called with a **String value "Hello"**.
* The **X type is inferred** as String.
* It returns the same **String value "Hello"**.

**🔧 Step 2: Call doSomething(7)**

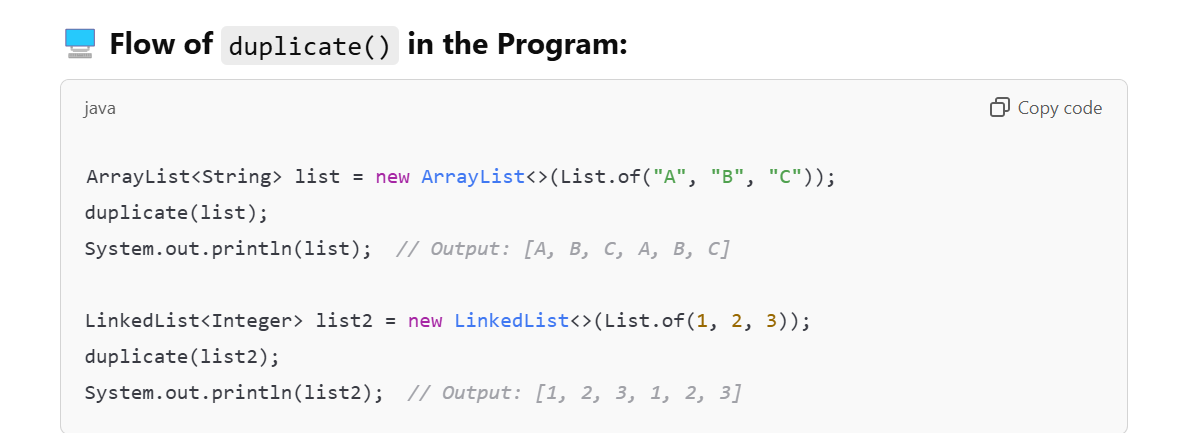
* The method is called with an **Integer value 7**.
* The **X type is inferred** as Integer.
* It returns the same **Integer value 7**.





**💡 What It Does:**

* This is a **generic method** that accepts a parameter of type **X**, where **X must extend List**.
* It **duplicates the elements in the list** by calling **addAll()**, which adds all elements of the list to itself.

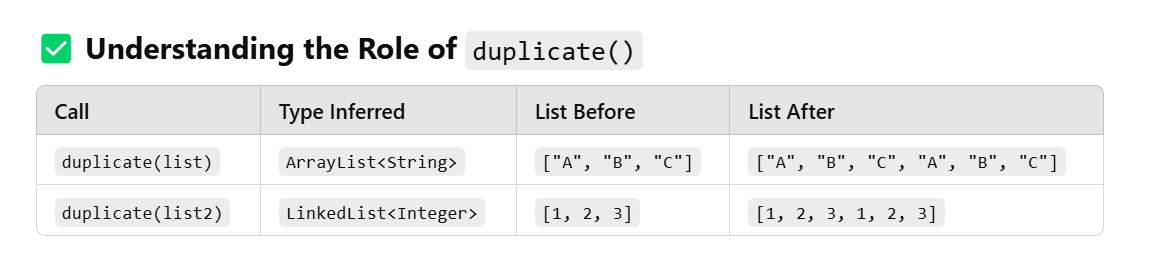


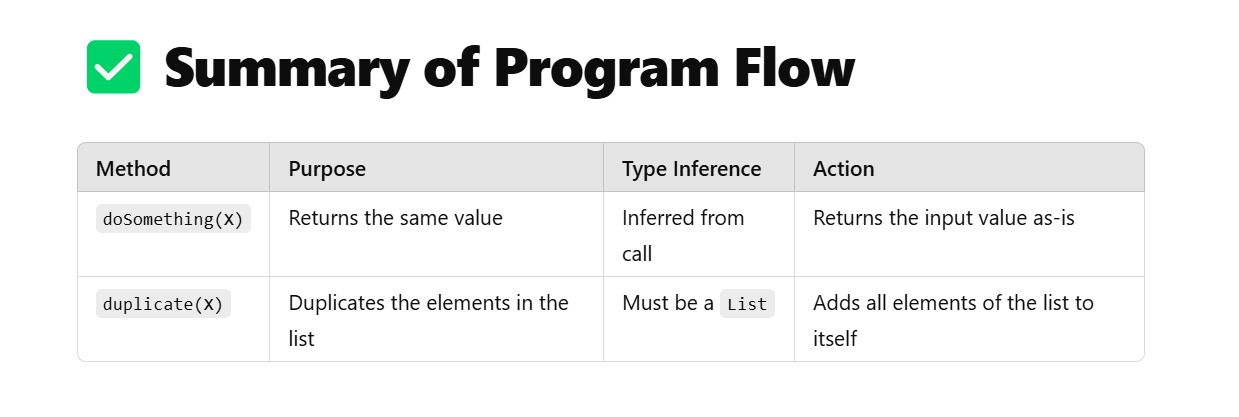
**🔧 Step 1: Call duplicate(list) with ArrayList<String>**

* The method is called with an **ArrayList containing ["A", "B", "C"]**.
* **X is inferred as ArrayList<String>**.
* **list.addAll(list)** duplicates the elements in the list.
* After the method call, the list becomes:  
  ["A", "B", "C", "A", "B", "C"].

**🔧 Step 2: Call duplicate(list2) with LinkedList<Integer>**

* The method is called with a **LinkedList containing [1, 2, 3]**.
* **X is inferred as LinkedList<Integer>**.
* **list2.addAll(list2)** duplicates the elements in the list.
* After the method call, the list becomes:  
  [1, 2, 3, 1, 2, 3].





**📝 Why Use Generic Methods?**

1. **Type Safety:**  
   Ensures that only the correct types are used.
2. **Code Reusability:**  
   Allows you to write **one method** that works for **multiple types**.

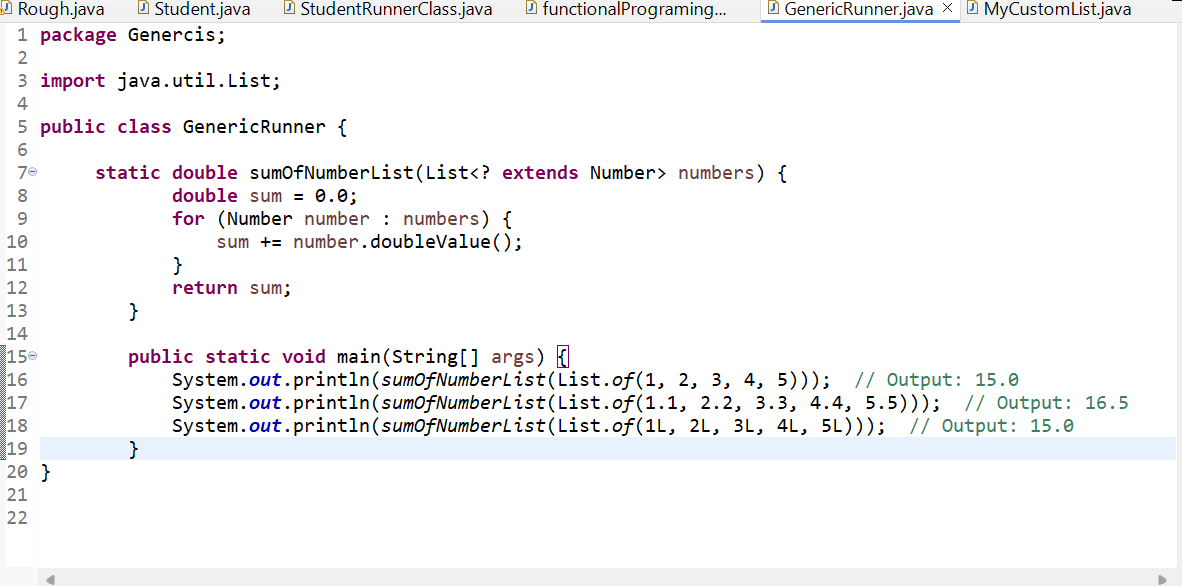
**✅ Generics and Wildcards**

**Wildcards** in generics allow more **flexibility** when working with unknown types.

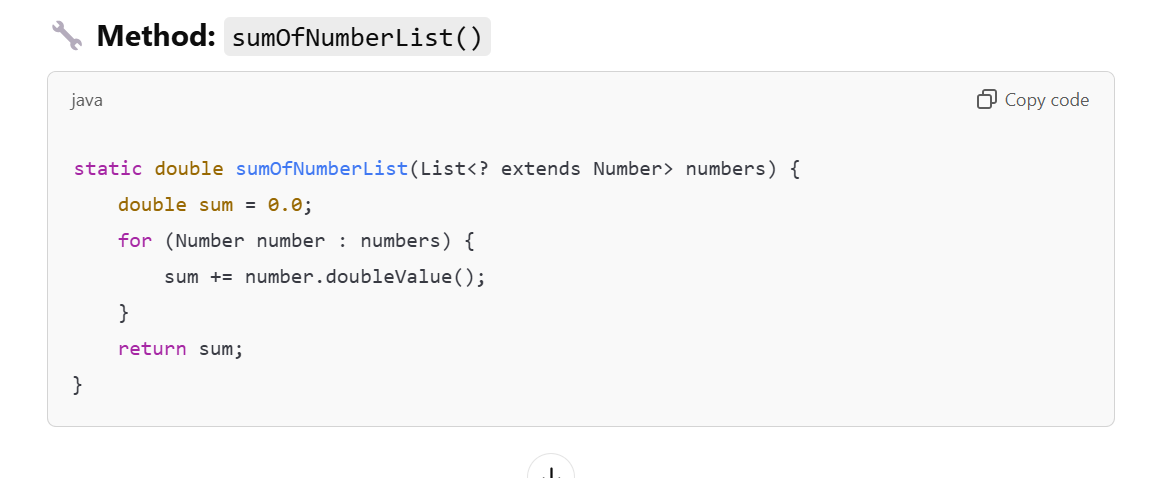
There are two main types of wildcards:

1. **Upper-Bounded Wildcard (? extends Type)**
2. **Lower-Bounded Wildcard (? super Type)**

🖥️ **Snippet 3: Upper-Bounded Wildcard Example**



✅ **Program Overview**

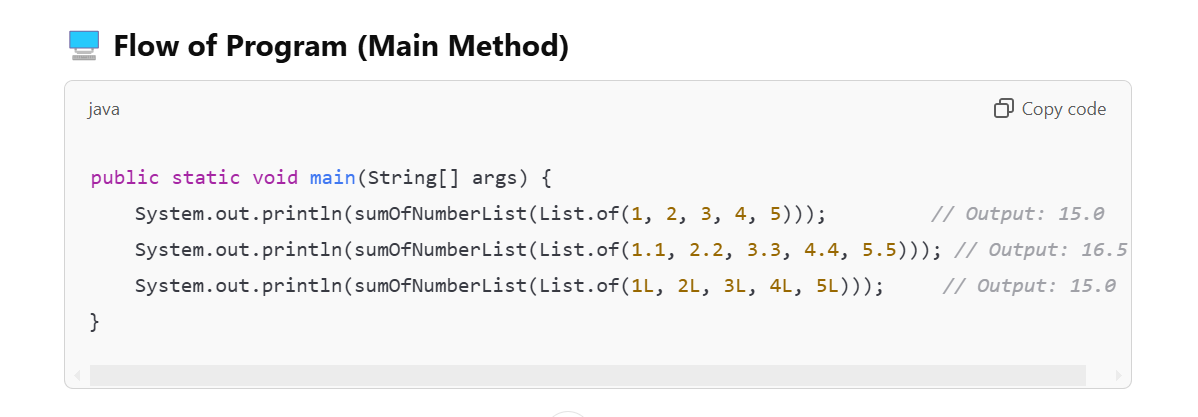


**📘 What is ? extends Number?**

* **?** is a **wildcard** that represents an **unknown type**.
* **? extends Number** means that the method can accept a **List of any type that is a subclass of Number**.

This includes:

* Integer
* Double
* Long
* Float
* Short
* Byte

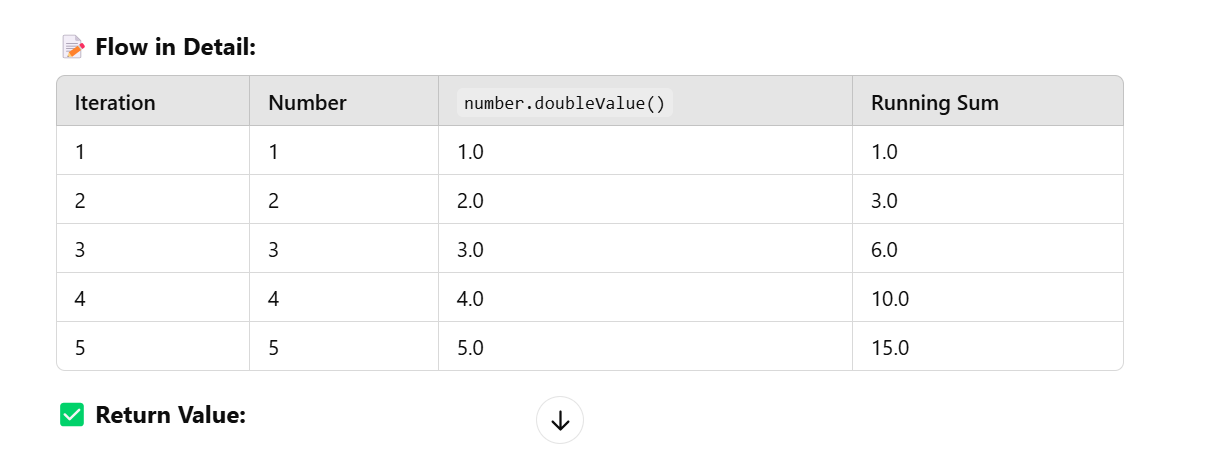


**✅ Step-by-Step Flow of Method Calls**

**🔧 Call 1: sumOfNumberList(List.of(1, 2, 3, 4, 5))**

**🔍 What's Happening?**

1. The method is called with a **List of Integers**: [1, 2, 3, 4, 5].
2. **? extends Number** allows the method to accept a **List<Integer>**.
3. The method initializes sum = 0.0.
4. It iterates over each element in the list:
   * **number.doubleValue()** converts each Integer to double.
   * The sum is updated after each iteration.



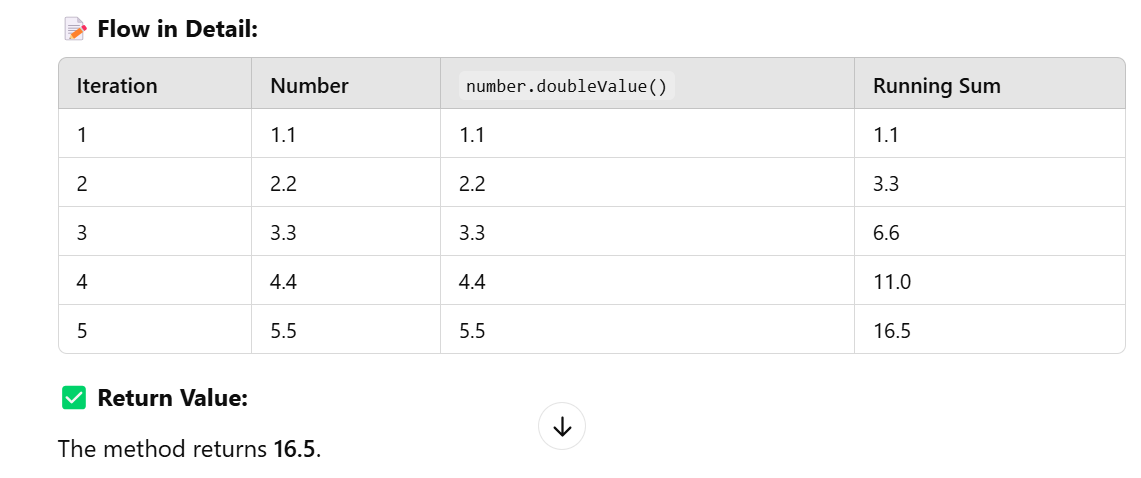
**✅ Return Value:**

The method returns **15.0**.

**🔧 Call 2: sumOfNumberList(List.of(1.1, 2.2, 3.3, 4.4, 5.5))**

**🔍 What's Happening?**

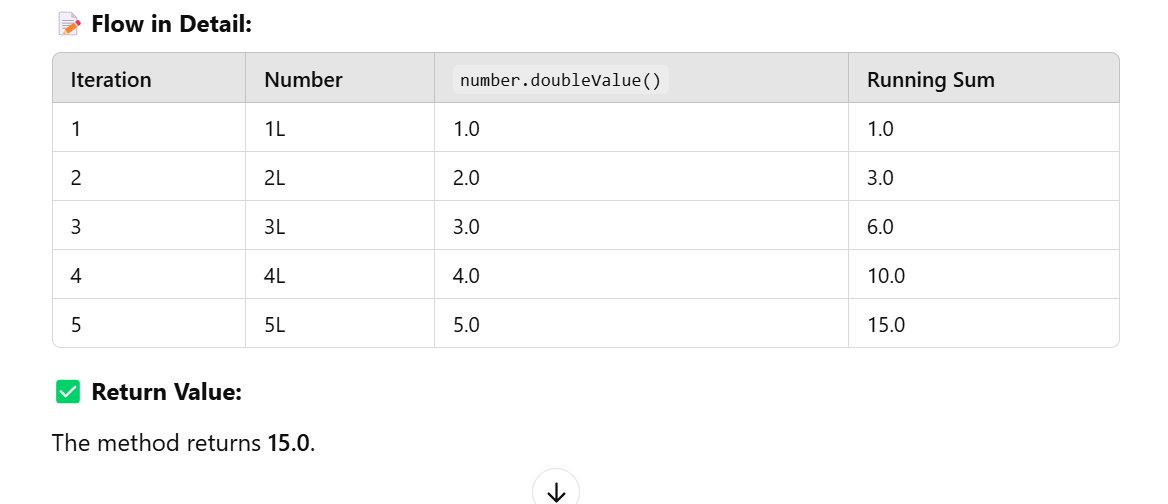
1. The method is called with a **List of Doubles**: [1.1, 2.2, 3.3, 4.4, 5.5].
2. **? extends Number** allows the method to accept a **List<Double>**.
3. The method initializes sum = 0.0.
4. It iterates over each element in the list:
   * **number.doubleValue()** is already a double.
   * The sum is updated after each iteration.

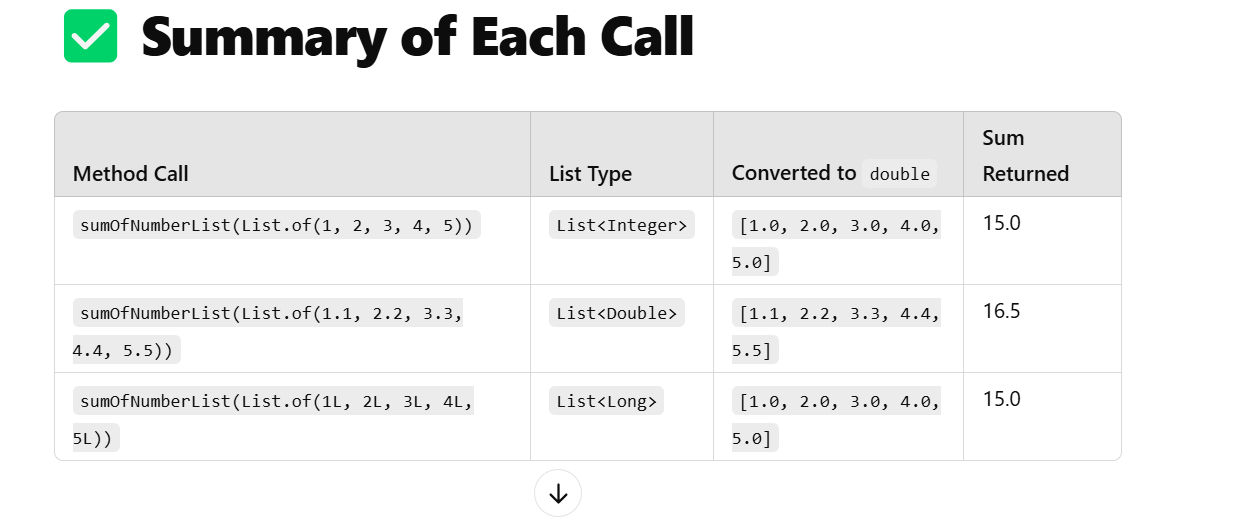


**🔧 Call 3: sumOfNumberList(List.of(1L, 2L, 3L, 4L, 5L))**

**🔍 What's Happening?**

1. The method is called with a **List of Longs**: [1L, 2L, 3L, 4L, 5L].
2. **? extends Number** allows the method to accept a **List<Long>**.
3. The method initializes sum = 0.0.
4. It iterates over each element in the list:
   * **number.doubleValue()** converts each Long to double.
   * The sum is updated after each iteration.

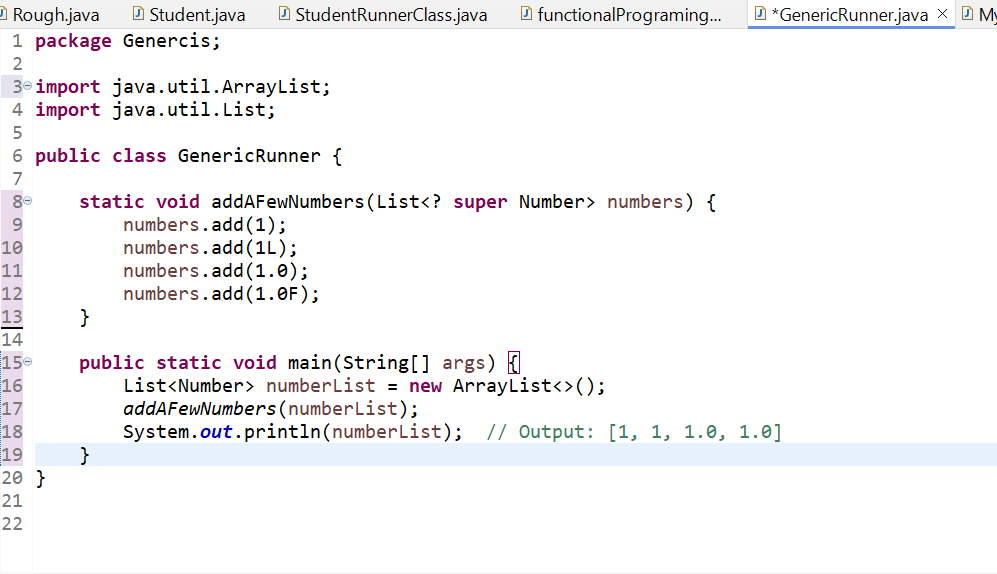




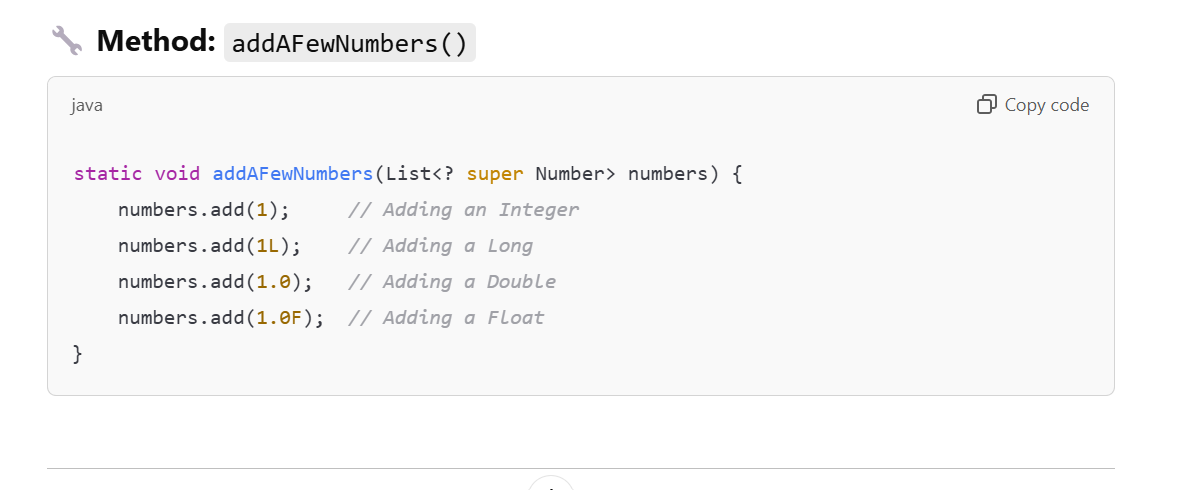
**✅ Why Use Wildcards?**

* **Upper-bounded wildcards (? extends Number)**:  
  Ensures that the method accepts **only numeric lists** (Integer, Double, Long, etc.) while keeping it **flexible** to handle different types.

🖥️ **Snippet 4: Lower-Bounded Wildcard Example**

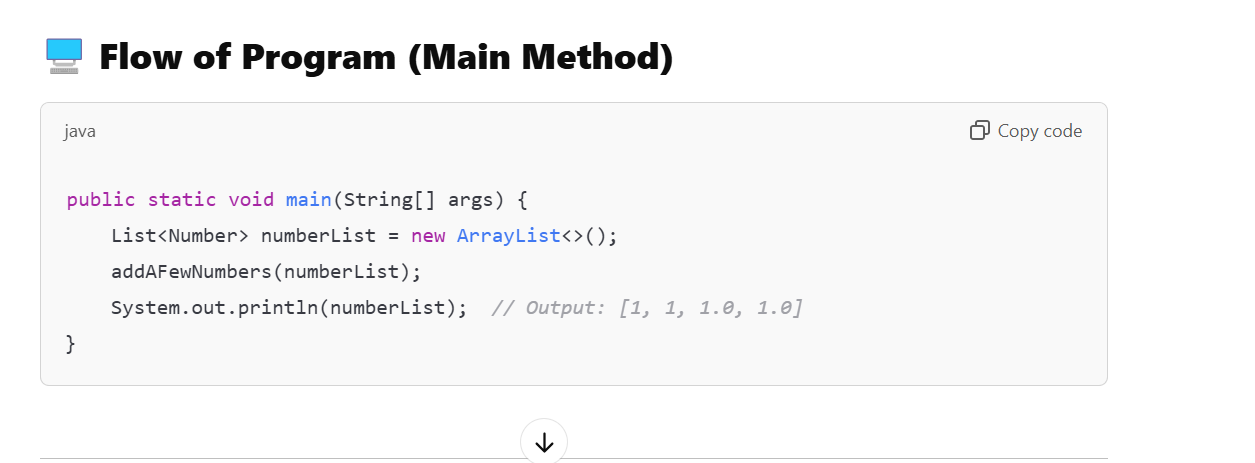
****

**✅ Understanding the Program with Lower-Bounded Wildcards (? super Number)**

****

**📘 What is ? super Number?**

* **?** is a **wildcard** that represents an **unknown type**.
* **? super Number** means that the method can accept a **List of any type that is a superclass of Number**.
* This includes:
  + **Number**
  + **Object** (since Object is the superclass of all classes in Java)



**✅ Step-by-Step Flow of Method Calls**

**🔧 Step 1: Creating a List of Numbers**

java

Copy code

List<Number> numberList = new ArrayList<>();

* A **List of Number** is created.
* It is passed to the **addAFewNumbers()** method.

**🔧 Step 2: Calling addAFewNumbers(numberList)**

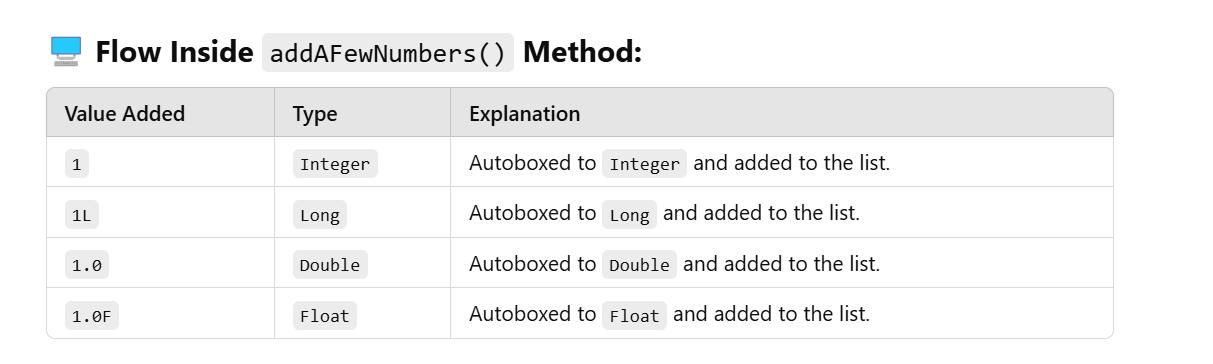
java

Copy code

addAFewNumbers(numberList);

**🔍 What's Happening?**

1. The method accepts a **List** with a type parameter that is a **superclass of Number**.
2. The method **adds different types of numbers** to the list:
   * **1 (Integer)**
   * **1L (Long)**
   * **1.0 (Double)**
   * **1.0F (Float)**



**🔧 Step 3: Printing the List**

java

Copy code

System.out.println(numberList);

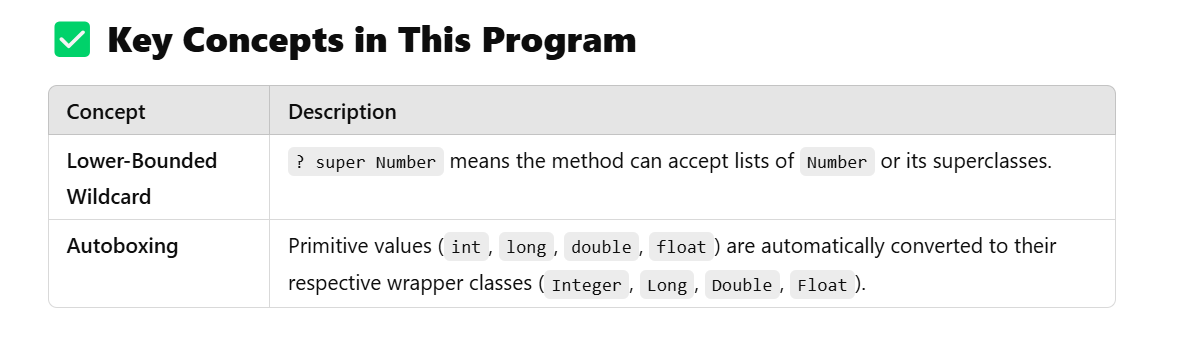
**🔍 What’s in the List?**

The list now contains:

csharp

Copy code

[1, 1, 1.0, 1.0]



**✅ Why Use Lower-Bounded Wildcards?**

1. **Allows Flexibility**:  
   The method can accept **a list of Number or any of its superclasses**.
2. **Type Safety**:  
   You can **add specific types** (Integer, Long, Double, Float) to the list without type mismatch issues.

**📝 What Happens if We Change the List Type?**

1. If we pass a **List<Object>** instead of **List<Number>**, the method will still work because **Object** is a **superclass** of **Number**.

