

# Default method in interface



explain default method in interface

Okay, let's go step by step.

## 1. Normal interface methods (before Java 8)

Traditionally, all methods inside an interface were **abstract** (no body).

Example:

```
java

interface Vehicle {
    void start();    // abstract
    void stop();     // abstract
}
```

→ Any class implementing `Vehicle` **must** give body to both methods.

## 2. Problem before Java 8

Suppose you already have 50 classes implementing `Vehicle`.

Now you want to add a new method `honk()` to the interface.

If methods are abstract, then **all 50 classes break**, because they must implement the new method.

## 3. Solution → Default Method (Java 8 onwards)

Java introduced **default methods in interfaces**.

They allow you to provide a **method body inside the interface itself**.

Syntax:

```
java

interface Vehicle {
    void start();
    void stop();

    default void honk() {
        System.out.println("Beep Beep!");
    }
}
```

→ Now, if an implementing class doesn't override `honk()`, it will **automatically get the default implementation**.

→ But if needed, a class can still **override** it.

## 4. Example

```
java

interface Vehicle {
    void start();
    void stop();

    default void honk() {
        System.out.println("Beep Beep!");
    }
}

class Car implements Vehicle {
    public void start() {
        System.out.println("Car starting...");
    }
    public void stop() {
        System.out.println("Car stopping...");
    }
    // no need to write honk(), still works
}

class Bike implements Vehicle {
    public void start() {
        System.out.println("Bike starting...");
    }
    public void stop() {
        System.out.println("Bike stopping...");
    }
    // override default
    public void honk() {
        System.out.println("Bike horn: Peeep!");
    }
}

public class Main {
    public static void main(String[] args) {
        Vehicle car = new Car();
        car.start();
        car.honk();
        car.stop();

        Vehicle bike = new Bike();
        bike.start();
        bike.honk();
        bike.stop();
    }
}
```

### Output:

```
mathematica

Car starting...
Beep Beep!
Car stopping...

Bike starting...
Bike horn: Peeep!
Bike stopping...
```

## 5. When to use Default Methods?

- When you want to **add new functionality** to interfaces **without breaking existing classes**.

- To provide a **common implementation** that can still be **overridden if needed**.

👉 In one line:

**A default method in an interface is a method with a body that implementing classes can either use as-is or override.**

Do you want me to also explain **how default method is different from static method in interface** (since both have bodies)?



How to HashSet and HashMap work internally in java ? what is the role of equals() and hashCode() methods in context ?

explain and then give me short summarized answer

Great question 👍 Let's break it down step by step.

## 1. HashSet internally

- **HashSet** is just a wrapper around a **HashMap**.
- When you add an element to a **HashSet**, it is actually stored as a **key** in a **HashMap**. (The value is a constant dummy object.)

Example:

```
java

HashSet<String> set = new HashSet<>();
set.add("apple");
```

Internally:

```
java

map.put("apple", DUMMY);
```

So, the uniqueness of **HashSet** comes from how **HashMap** stores keys.

## 2. HashMap internally

- A **HashMap** stores data in **buckets** (an array of linked lists / tree nodes after Java 8).
- When you insert a key:
  1. **hashCode()** of the key is called → used to find bucket index.
  2. If the bucket is empty → store entry.

3. If bucket already has entries → `equals()` is used to check:

- If the key already exists → update value.
- Else → add new entry to that bucket (chain/tree).

### 3. Role of `hashCode()` and `equals()`

- `hashCode()` → Decides *which bucket* an object goes to.
- `equals()` → Decides *if two objects are actually equal* when they land in the same bucket.

🔑 **Both must follow contract:**

- If `a.equals(b)` is true → both must have same `hashCode()`.
- If two objects have same `hashCode()`, they *may* or *may not* be equal (`equals()` decides).

### 4. Example

```
java

class Student {
    int id;
    String name;

    public int hashCode() {
        return id; // bucket decided by id
    }
    public boolean equals(Object o) {
        if(this == o) return true;
        if(o == null || getClass() != o.getClass()) return false;
        Student s = (Student) o;
        return id == s.id && name.equals(s.name);
    }
}
```

If two students have same `id` & `name`, `HashSet` will treat them as duplicates.

### 🔥 Short summarized answer

- **HashSet** uses a **HashMap** internally, storing elements as keys.
- **HashMap** uses `hashCode()` to find the bucket, and `equals()` to check duplicates within the bucket.
- **Contract:**
  - If `a.equals(b)` → then `a.hashCode() == b.hashCode()`.
  - If `hashCode` is same → `equals()` is used to confirm equality.

👉 Without correct `hashCode()` and `equals()`, `HashSet/HashMap` may behave unpredictably (e.g., duplicates or missing elements).

Do you want me to also **draw a diagram of how HashMap buckets and collisions work** (linked list/tree inside buckets)?



both internally uses hashtable right so give answer according to hashtable how equals and hashCode works

Perfect 🍌 Let's explain it directly in terms of **Hashtable-like working** (since both **HashMap** and **HashSet** are essentially built on the *hash table* data structure).

## 🔍 How Hash Table works internally

A **hash table** is basically:

- An **array of buckets**.
- Each bucket can hold multiple entries (linked list / tree).
- A **hash function** decides which bucket an entry goes into.

When you put or get an object:

1. Call `hashCode()` → generates an integer hash.
2. Calculate **bucket index**: `index = hashCode % array.length`.
3. If bucket empty → insert directly.
4. If bucket not empty (collision) → compare with existing entries using `equals()`:
  - If `equals()` returns `true` → treat as same key (update value / reject duplicate in HashSet).
  - Else → add as a new entry in the same bucket chain.

## 🎯 Role of `hashCode()` and `equals()` in Hashtable

- `hashCode()`
  - Decides the *bucket location* for an object.
  - If two objects have different hash codes → they go into different buckets (no equals check needed).
  - If two objects have the same hash code → they go into the *same bucket* → now `equals()` is checked.
- `equals()`
  - Decides if two objects in the same bucket are really *equal*.
  - Prevents duplicates (in HashSet) or ensures key uniqueness (in HashMap).

## ✅ Example (Hashtable perspective)

```
java
```

```
HashSet<String> set = new HashSet<>();
set.add("cat");
set.add("dog");
set.add("cat"); // duplicate
```

Internally:

- `"cat".hashCode()` → say 98273 → goes to bucket[5].
- `"dog".hashCode()` → say 30012 → goes to bucket[2].
- `"cat".hashCode()` again → goes to bucket[5].
- Now HashSet checks `"cat".equals("cat")` → true → so it rejects the duplicate.

## Short summarized answer (Hashtable view)

- **HashSet/HashMap** use a **hashtable** internally.
- `hashCode()` → decides the bucket where object is stored.
- `equals()` → used when multiple objects land in the same bucket, to check if they are actually the same.
- **Contract:**
  - If two objects are equal (`equals() == true`) → they must have the same `hashCode()`.
  - But same `hashCode()` doesn't guarantee equality → `equals()` must confirm.

👉 In short: `hashCode()` chooses the *bucket*, `equals()` decides the *exact match* inside that bucket.

Do you want me to also show you a **step-by-step diagram of bucket + linked list** for collisions (visual of how objects are stored)?



what is the initial capacity of a collection and what happens when it is exhausted ? Explain with examples

Great question 👍 Let's break this into two parts:

## 1. Initial Capacity in Java Collections

- **Capacity** = number of elements a collection (like `ArrayList`, `HashMap`, etc.) can hold **before resizing**.
- It is **not** the same as *size* (number of actual elements present).
- Different collections have different default capacities.

Examples:

- `ArrayList` → default capacity is **10**.
- `HashMap` → default capacity is **16** (must be power of 2).
- `HashSet` → internally uses `HashMap`, so default capacity = **16**.

## 2. What happens when capacity is exhausted?

When a collection runs out of capacity:

1. It **creates a new bigger array/table**.
2. **Copies old elements** into the new one.
3. Adds the new element.

The growth factor differs by collection:

- **`ArrayList`:**
  - New capacity = `oldCapacity * 1.5` (approx).
  - Example: 10 → 15 → 22 → 33 ...
- **`HashMap` / `HashSet`:**
  - New capacity = `oldCapacity * 2`.
  - Example: 16 → 32 → 64 ...
  - Also triggers **rehashing** (recalculating bucket positions for all keys).

## 3. Examples

### Example 1: `ArrayList`

```
java

ArrayList<Integer> list = new ArrayList<>();
for(int i=1; i<=12; i++) {
    list.add(i);
    System.out.println("Added: " + i + " | Size: " + list.size());
}
```

- First 10 elements → fit in initial capacity (10).
- On adding 11th element → `ArrayList` resizes to 15 internally.
- No error, resizing is automatic.

### Example 2: `HashMap`

```
java

HashMap<Integer, String> map = new HashMap<>();
for(int i=1; i<=20; i++) {
    map.put(i, "val" + i);
}
```

```
System.out.println("Added key " + i);
}
```

- Default capacity = 16, load factor = 0.75.
- Resize threshold =  $16 * 0.75 = 12$ .
- When 13th element is inserted → HashMap resizes capacity to 32, and rehashes all keys.

## Short summarized answer

- **Initial capacity** = how many elements a collection can hold before needing to grow.
- When exhausted:
  - **ArrayList** → grows 1.5x.
  - **HashMap/HashSet** → doubles capacity & rehashes.
- Growth is **automatic** → you don't get errors, but resizing has performance cost (copying & rehashing).

Would you like me to also make a **comparison table of default capacities + growth policies** for common collections (**ArrayList**, **Vector**, **HashMap**, **HashSet**, etc.)?



Is HashMap threadsafe ? How to make it thread safe ?

Let's carefully go step by step.

## 1. Is HashMap thread-safe?

- **No. HashMap is not thread-safe.**
- If multiple threads access a **HashMap** concurrently **and at least one thread modifies it**, you can get:
  - Data corruption
  - Infinite loops during rehashing
  - Unexpected results

### Example (unsafe):

```
java

Map<Integer, String> map = new HashMap<>();

// Thread 1 adds elements
// Thread 2 iterates elements
// Concurrent modification can cause errors
```



## 2. How to make HashMap thread-safe?

There are several approaches:

### a) SynchronizedMap

- Wrap HashMap using `Collections.synchronizedMap()`:

```
java
Map<Integer, String> map = Collections.synchronizedMap(new HashMap<>());
```

- Ensures **all method calls are synchronized**.
- **Caution:**  
Iterating still requires manual synchronization:

```
java
synchronized(map) {
    for(Map.Entry<Integer,String> e : map.entrySet()){
        System.out.println(e.getKey() + " " + e.getValue());
    }
}
```

### b) ConcurrentHashMap (Preferred)

- Introduced in Java 1.5, designed for concurrency.
- Allows **multiple threads to read/write concurrently**.
- Uses **bucket-level locking / striped locking** → better performance than synchronizedMap.

```
java
Map<Integer, String> map = new ConcurrentHashMap<>();
map.put(1, "One");
map.put(2, "Two");
```

- No need to manually synchronize during iteration.
- Iterators are **weakly consistent**: they may reflect some updates while iterating.

### c) Using synchronized blocks (manual)

```
java
Map<Integer, String> map = new HashMap<>();
synchronized(map) {
    map.put(1, "One");
}
```

- Less convenient, prone to errors if not careful.

## 🔥 Short summarized answer

- **HashMap is not thread-safe.**
- **Thread-safe options:**
  1. `Collections.synchronizedMap(new HashMap<>())` → full synchronization, slower iteration.
  2. `ConcurrentHashMap` → concurrent access with better performance, preferred in modern apps.

If you want, I can also **draw a diagram showing difference between HashMap, synchronizedMap, and ConcurrentHashMap** concurrency behavior—it makes it very clear.

Do you want me to do that?



what is failfast and failsafe iterator ? explain with code

Sure! Let's go step by step so it's crystal clear.

## 1. Fail-Fast Iterator

- **Definition:** An iterator that **throws** `ConcurrentModificationException` if the collection is **modified structurally** while iterating.
- Structural modification = adding/removing elements (not just changing values).
- Most **Collection Framework classes** like `ArrayList`, `HashSet`, `HashMap` provide **fail-fast iterators**.

**Example (Fail-Fast with ArrayList):**

```
java

import java.util.*;

public class FailFastExample {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        Iterator<String> it = list.iterator();
        while(it.hasNext()) {
            String val = it.next();
            if(val.equals("B")) {
                list.remove(val); // Structural modification
            }
        }
    }
}
```

**Output:**

```
cpp
```

```
Exception in thread "main" java.util.ConcurrentModificationException
```

### ✓ Key points:

- Detects modification **immediately** during iteration.
- Works on **ArrayList, HashSet, HashMap** (fail-fast iterators).
- Faster because it **doesn't copy the collection**.

## 2. Fail-Safe Iterator

- **Definition:** An iterator that **does NOT throw exception** if the collection is modified during iteration.
- Usually works on **Concurrent collections** like **CopyOnWriteArrayList, ConcurrentHashMap**.
- Iterates over a **snapshot** of the collection (copy of data).

### Example (Fail-Safe with CopyOnWriteArrayList):

```
java

import java.util.*;
import java.util.concurrent.CopyOnWriteArrayList;

public class FailSafeExample {
    public static void main(String[] args) {
        List<String> list = new CopyOnWriteArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        Iterator<String> it = list.iterator();
        while(it.hasNext()) {
            String val = it.next();
            if(val.equals("B")) {
                list.remove(val); // No exception
            }
            System.out.println(val);
        }

        System.out.println("Updated list: " + list);
    }
}
```

### Output:

```
less

A
B
C
Updated list: [A, C]
```

### ✓ Key points:

- Does **not throw exception**.

- Slower because it **creates a copy** of the collection.
- Works on **CopyOnWriteArrayList**, **ConcurrentHashMap**, **ConcurrentSkipListMap**.

### 3. Comparison Table

Feature	Fail-Fast	Fail-Safe
Throws Exception	Yes ( <b>ConcurrentModificationException</b> )	No
Collection Type	Non-concurrent ( <b>ArrayList</b> , <b>HashMap</b> )	Concurrent ( <b>CopyOnWriteArrayList</b> , <b>ConcurrentHashMap</b> )
Iteration Mechanism	Directly on collection	On a <b>clone / snapshot</b>
Performance	Fast	Slower (due to copying)

#### ✓ Short Summary

- **Fail-Fast:** Detects concurrent modification → throws exception.
- **Fail-Safe:** Ignores concurrent modification → safe iteration using a snapshot.

If you want, I can also **draw a diagram showing how Fail-Fast vs Fail-Safe iterators work internally**, it makes it super clear visually.

Do you want me to do that?



what is lambda expression in java ? how datatypes of lambda arguments and return types are determined ?

Let's go step by step.

## 1. What is a Lambda Expression?

- **Lambda expression** = a short way to represent an **anonymous function** (a function without a name) in Java.
- Introduced in **Java 8**.
- Makes it easier to write **functional-style code** (like in streams).

**Syntax:**

```
java
(parameters) -> expression
```

or

```
java

(parameters) -> { statements; }
```

### Example:

```
java

// Using Runnable (functional interface)
Runnable r = () -> System.out.println("Hello World!");
r.run();
```

- Here `() -> System.out.println("Hello World!");` is a **lambda expression**.

## 2. Functional Interface

- Lambda expressions can **only be used with interfaces that have exactly one abstract method**.
- These are called **Functional Interfaces**.
- Examples: `Runnable`, `Comparator`, `Predicate`, `Function`.

### Example with Comparator:

```
java

List<String> names = Arrays.asList("John", "Alice", "Bob");

// Sort using lambda
Collections.sort(names, (a, b) -> a.compareTo(b));

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

- `(a, b) -> a.compareTo(b)` is the lambda expression.
- The compiler knows **types of a and b** from the `Comparator<String>` interface.

## 3. How data types of lambda arguments and return type are determined

- Java compiler uses **"Target Typing"**.
- The **context** where the lambda is used decides:
  - Argument types
  - Return type

### Examples

#### 1. Type inferred from functional interface

```
java

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

// Lambda assigned to Predicate<Integer>
Predicate<Integer> isEven = n -> n % 2 == 0;
```

- `n` is automatically inferred as `Integer` from `Predicate<Integer>`.
- Return type is `boolean` (matches `test()` method of `Predicate`).

## 2. Explicit types (optional)

```
java

BinaryOperator<Integer> sum = (Integer a, Integer b) -> a + b;
```

- Here `a` and `b` types are explicitly specified.
- Return type inferred from `a + b` expression  $\rightarrow$  `Integer`.

# 4. Rules

1. Number of parameters in lambda must match the functional interface method.
2. Return type must match the functional interface method.
3. Types can be omitted if compiler can infer them.

## ✓ Short Summary

- **Lambda expression:** anonymous function represented as `(parameters) -> expression` in Java.
- **Data types of arguments:** inferred from the **target functional interface**.
- **Return type:** inferred from the **functional interface's abstract method**.

If you want, I can also make a **diagram showing how lambda maps to functional interface method internally**, which makes it very clear visually.

Do you want me to do that?



lambda expression only used in functional interfaces ?

Yes, **lambda expressions in Java can only be used with functional interfaces**. Let me explain carefully.

## 1. Functional Interface

- A **functional interface** is an interface with **exactly one abstract method**.
- Can have **default** or **static** methods, but only **one abstract method** counts.
- Lambda expressions provide the **implementation** for that single abstract method.

### Examples:

```
java
@FunctionalInterface
interface MyFunction {
    void execute();
}
```

- The `execute()` method is the **only abstract method**, so this is valid.

## 2. Why only functional interfaces?

- A lambda is essentially a **short way to implement a method**.
- If the interface has **more than one abstract method**, compiler **doesn't know which method** the lambda is for → **compilation error**.

### Example (Invalid):

```
java
interface NotFunctional {
    void method1();
    void method2();
}

// Cannot use lambda because 2 abstract methods exist
NotFunctional nf = () -> System.out.println("Hello"); // ❌ Error
```

## 3. Compiler Role

- When you write a lambda like `(x) -> x*2`, the compiler checks the **context**:
  - Which interface is expected?
  - That interface must have **exactly one abstract method**.
- Then it generates an **implementation** of that method behind the scenes (as an anonymous class or invokedynamic).

## 4. Summary

Feature	Lambda Expression Rule
Can assign to	Only <b>functional interface</b>
Number of abstract methods allowed	1

Feature	Lambda Expression Rule
Default/static methods	Allowed in interface, do <b>not affect</b> lambda usage
Compiler	Infers lambda's parameters and return type from the functional interface

### ✓ In short:

**Yes, lambda expressions can only be used with functional interfaces**, because they provide the implementation for the **single abstract method**.

If you want, I can also give a **list of common functional interfaces in Java** and show how lambdas are used with them.

Do you want me to do that?



What is a functional interface ? which functional interfaces are predefined and where they are used ?

consider only these functional interfaces

Predicate  
consumer  
bifunction  
unary operator  
binary operator

Perfect 👍 Let's deep dive into **functional interfaces** and then focus only on the ones you listed.

## 1. What is a Functional Interface?

- A **functional interface** is an interface with **exactly one abstract method**.
- Lambdas or method references can provide implementation for that single method.
- Marked with `@FunctionalInterface` annotation (not mandatory but good practice).

### Example:

```
java
@FunctionalInterface
interface MyFunc {
    void doSomething();
}
```

➡ Can be used as:



```
java

MyFunc f = () -> System.out.println("Hello Functional Interface!");
```

## 2. Predefined Functional Interfaces (Java 8+)

Java provides many in `java.util.function` package.

You want me to focus on **Predicate**, **Consumer**, **BiFunction**, **UnaryOperator**, **BinaryOperator**. Let's go one by one.

### a) Predicate<T>

- **Definition:** Takes one argument and returns a boolean.
- **Method:**

```
java

boolean test(T t)
```

- **Use case:** Filtering, conditional checks.
- **Example:**

```
java

Predicate<Integer> isEven = n -> n % 2 == 0;
System.out.println(isEven.test(4)); // true
System.out.println(isEven.test(7)); // false
```

### b) Consumer<T>

- **Definition:** Takes one argument and returns nothing (void).
- **Method:**

```
java

void accept(T t)
```

- **Use case:** Printing, logging, modifying external state.
- **Example:**

```
java

Consumer<String> printer = s -> System.out.println("Hello, " + s);
printer.accept("Uday"); // prints "Hello, Uday"
```

### c) BiFunction<T,U,R>

- **Definition:** Takes two arguments and returns a result.
- **Method:**

```
java
R apply(T t, U u)
```

- **Use case:** Operations involving two inputs (like sum, concat).
- **Example:**

```
java
BiFunction<Integer, Integer, Integer> sum = (a, b) -> a + b;
System.out.println(sum.apply(5, 3)); // 8
```

## d) UnaryOperator<T>

- **Definition:** A special case of `Function<T,R>` where **input and output types are the same**.
- **Method:**

```
java
T apply(T t)
```

- **Use case:** Increment, square, string uppercase.
- **Example:**

```
java
UnaryOperator<Integer> square = n -> n * n;
System.out.println(square.apply(6)); // 36
```

## e) BinaryOperator<T>

- **Definition:** A special case of `BiFunction<T,U,R>` where **all three types are the same**.
- **Method:**

```
java
T apply(T t1, T t2)
```

- **Use case:** Add, multiply, min, max.
- **Example:**

```
java
BinaryOperator<Integer> multiply = (a, b) -> a * b;
System.out.println(multiply.apply(4, 5)); // 20
```

### 3. Short Comparison Table

Functional Interface	Parameters	Return Type	Use Case
<code>Predicate&lt;T&gt;</code>	1 (T)	boolean	Filtering, conditions
<code>Consumer&lt;T&gt;</code>	1 (T)	void	Printing, logging
<code>BiFunction&lt;T,U,R&gt;</code>	2 (T, U)	R	Operations with 2 inputs
<code>UnaryOperator&lt;T&gt;</code>	1 (T)	T	Transform one value
<code>BinaryOperator&lt;T&gt;</code>	2 (T, T)	T	Combine two values

#### Summary in one line:

A **functional interface** has exactly one abstract method; lambdas implement them.

- **Predicate** → test condition.
- **Consumer** → consume input (no return).
- **BiFunction** → two inputs, one result.
- **UnaryOperator** → one input/output same type.
- **BinaryOperator** → two inputs/output same type.

Do you want me to also show **how these five are used with Java Streams** (filter, map, reduce, forEach)? That would connect theory to real-world usage.



tell about function functional interface

Got it  Let's focus on the **Function functional interface** from `java.util.function`.

## 1. What is Function?

- A predefined **functional interface** introduced in **Java 8**.
- Represents a function that takes **one argument** and returns a **result**.

#### Definition:

```
java
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

- **T** → input type
- **R** → return type

## 2. Example

```
java

import java.util.function.Function;

public class FunctionExample {
    public static void main(String[] args) {
        Function<String, Integer> lengthFunction = str -> str.length();

        System.out.println(lengthFunction.apply("Uday"));    // 4
        System.out.println(lengthFunction.apply("Sunbeam")); // 7
    }
}
```

→ Takes a **String** as input and returns its **Integer** length.

## 3. Use Cases

- **Transforming values** (map operations in streams).
- **Type conversion** (e.g., **String** → **Integer**).
- **Business logic functions** (e.g., square of a number).

### Example in Streams:

```
java

List<String> names = List.of("Alice", "Bob", "Charlie");

names.stream()
    .map(str -> str.toUpperCase()) // Function<String,String>
    .forEach(System.out::println);
```

## 4. Default & Static Methods in Function

**Function** also provides methods for **function chaining**:

1. **andThen()** → apply one function, then another.

```
java

Function<Integer, Integer> square = x -> x * x;
Function<Integer, Integer> doubleValue = x -> x * 2;

Function<Integer, Integer> squareThenDouble = square.andThen(doubleValue);
System.out.println(squareThenDouble.apply(3)); // (3*3)=9, then 9*2=18
```

2. **compose()** → apply one function first, then another.

```
java
```

```
Function<Integer, Integer> squareThenDouble2 = doubleValue.compose(square);
System.out.println(squareThenDouble2.apply(3)); // square first → 9, then double → 18
```

3. **identity()** → returns a function that returns the input as is.

```
java
```

```
Function<String, String> identity = Function.identity();
System.out.println(identity.apply("Sunbeam")); // Sunbeam
```

## 5. Summary

- **Function<T,R>**: one input (T), one output (R).
- Abstract method: **R apply(T t)**.
- Used in **mapping, transformation, and type conversion**.
- Supports **chaining** via **andThen()**, **compose()**, and provides **identity()**.

👉 In short:

**Function is the most general functional interface for transforming one value into another.**

Do you also want me to explain how **Function** differs from **UnaryOperator** (since they look very similar)?