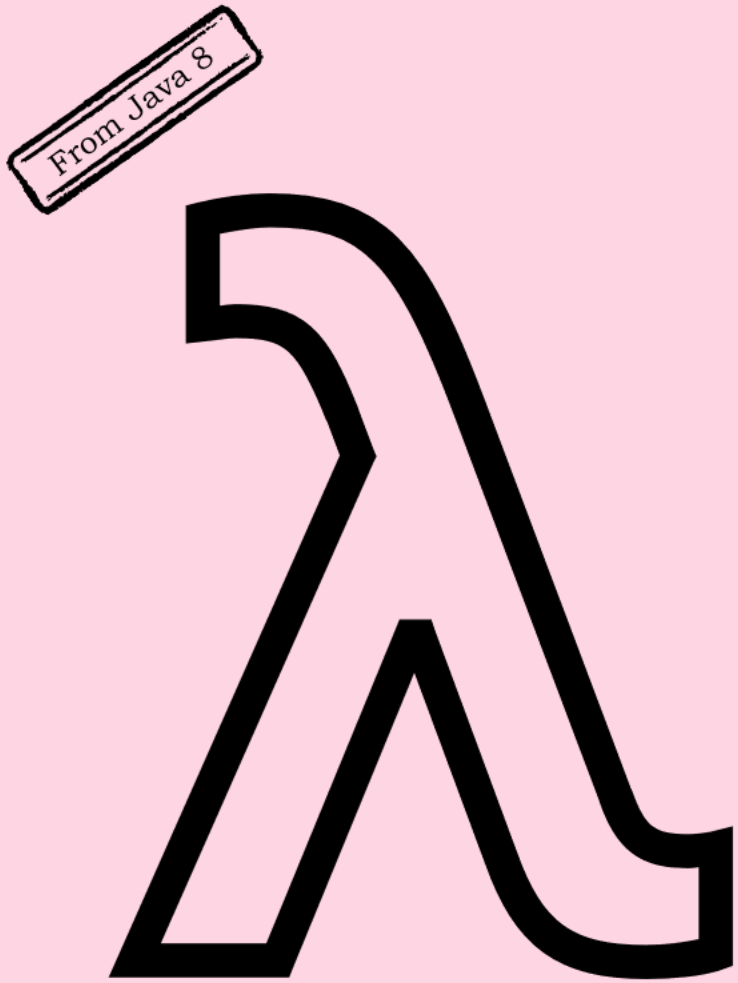
Functional Programming using lambda (λ) expressions

eazy bytes



Java has been changing fast, especially with Java 8. It's like a game-changer because it brought in **functional programming** support through **lambda expressions**. Before Java 8, we were stuck with writing code in an **imperative style**. But after Java 8, we got the option to write in either the old familiar imperative way or try out this new cool functional style. It's like having more tools in our coding toolbox!

The go-to method for writing code in Java has long been the imperative style, favored and widely used by developers. However, this familiar approach comes with its fair share of accidental complexities. Nowadays, an increasing number of developers are making the shift towards functional programming in Java. The appeal lies in the reduced complexity & enhanced readability of functional style code, especially once developers become accustomed to the syntax and paradigm.

Embracing the functional style in Java simplifies and streamlines our daily coding tasks, making them more straightforward and expressive. With this approach, we can effortlessly craft concise, elegant, and expressive code using fewer lines, thereby minimizing errors. Understanding the code becomes intuitive, eliminating the need to decipher intricate logic. The ultimate outcome is heightened productivity, enabling us to create and deliver applications at a faster pace.

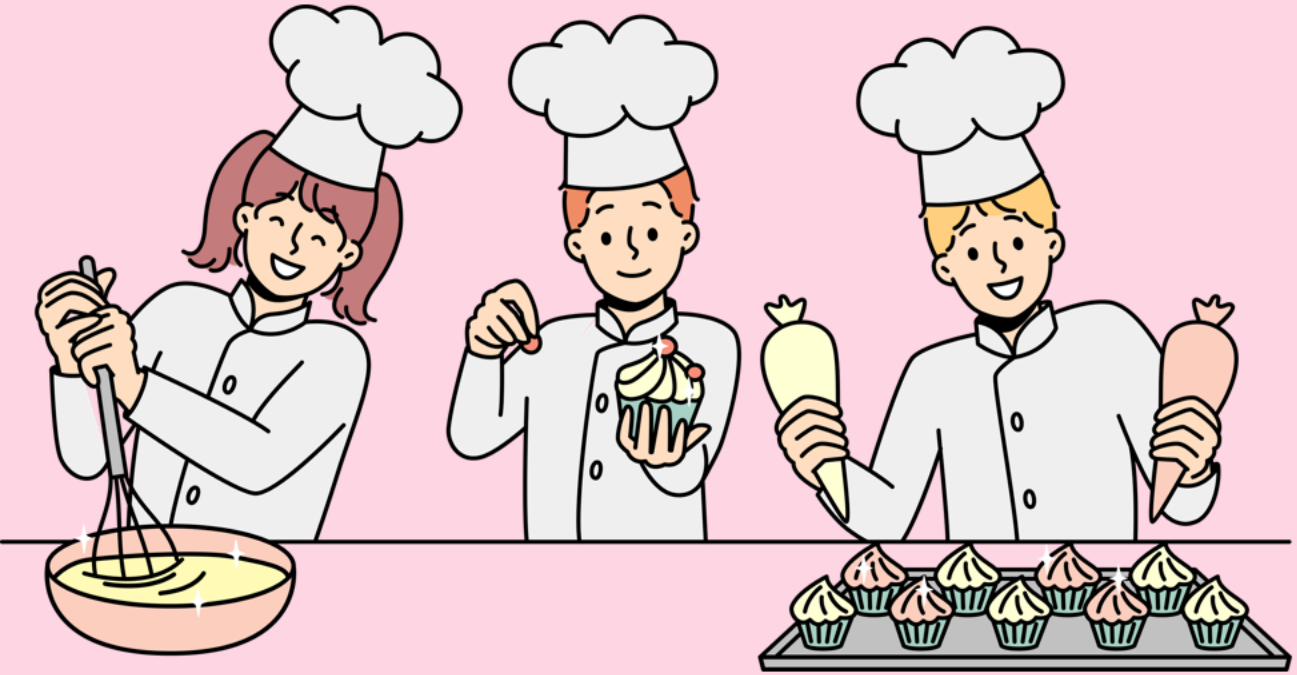
Imperative style vs functional style

eazy bytes

Imagine you're baking a cake:

Imperative Style (Procedural Programming):

- You're the head chef, giving step-by-step instructions (mix this, add that, bake for so long).
- You directly manipulate ingredients and equipment (beat eggs, pour batter, adjust oven temperature).
- Every detail of the process matters, and changing one step could alter the outcome.
- This style is familiar and often intuitive, but can be prone to errors and modifications might require rewriting parts of the recipe.



Functional Style (Functional Programming):

- You're the recipe author, describing the desired outcome (fluffy cake) and leaving the execution to expert bakers.
- You provide separate, specialized instructions for each step (mix dry ingredients, cream butter, etc.).
- Each step can be easily reused in other recipes (e.g., the "cream butter" function).
- This style emphasizes immutability (ingredients remain unchanged) and purity (no side effects like altering oven settings).
- It promotes modularity and testability, but can be less intuitive for beginners and may require different problem-solving approaches.

Imperative style vs functional style

eazy bytes

Applying to Java:

- Imperative Java focuses on loops, variables, and statements that modify state.
- Functional Java employs lambdas, streams, and immutability to create pure functions with no side effects.

Let's create examples in both imperative and functional styles for a common task: **finding the sum of squares of even numbers in a list**

Imperative Style:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        int sumOfSquaresOfEvens = 0;
        for (int number : numbers) {
            if (number % 2 == 0) {
                int square = number * number;
                sumOfSquaresOfEvens += square;
            }
        }

System.out.println("Sum of squares of even numbers (Imperative): " + sumOfSquaresOfEvens);
```

Imperative style vs functional style

eazy bytes

Functional Style:

```
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

int sumOfSquaresOfEvens = numbers.stream()
                .filter(number -> number % 2 == 0)
                .mapToInt(number -> number * number)
                .sum();

System.out.println("Sum of squares of even numbers (Functional): " + sumOfSquaresOfEvens);
```

The imperative style gets the job done, but it can feel a bit messy. It's not our fault; we worked with what we had. However, the code is quite low-level.

There's room for improvement—a lot of improvement. We can make our code look more like the requirement specification, closing the gap between what the business needs and the code that fulfills those needs. This reduces the chance of misinterpreting requirements.

Imperative style vs functional style

eazy bytes

In the functional style, the code is concise, but we're bringing in some new features from modern Java. We kick off by using the stream method on the numbers list. This opens the door to a special iterator with plenty of convenient functions.

Rather than explicitly going through the numbers list, we use special methods like filter and mapToInt. These methods take an anonymous function—a lambda expression—as a parameter, snugly placed within the parentheses (). Then, we call the sum method, a special form of the reduce operation, to compute the total based on the result of mapToInt, itself a special form of the map method.

Imperative style vs functional style

eazy bytes

Several enhancements are evident when comparing the functional approach to the imperative one:

1. Unlike the imperative style, functional programming liberates us from dealing with low-level operations. The code focuses on expressing what needs to be done rather than detailing how to do it.

2. Instead of manually iterating through data structures, functional programming leverages a set of library methods. This allows for more expressive and concise code, reducing the likelihood of errors.

3. Functional programming facilitates easier parallelization of tasks. The nature of functional code, with its emphasis on immutability and lack of shared state, aligns well with parallel processing paradigms, providing opportunities for performance improvements.

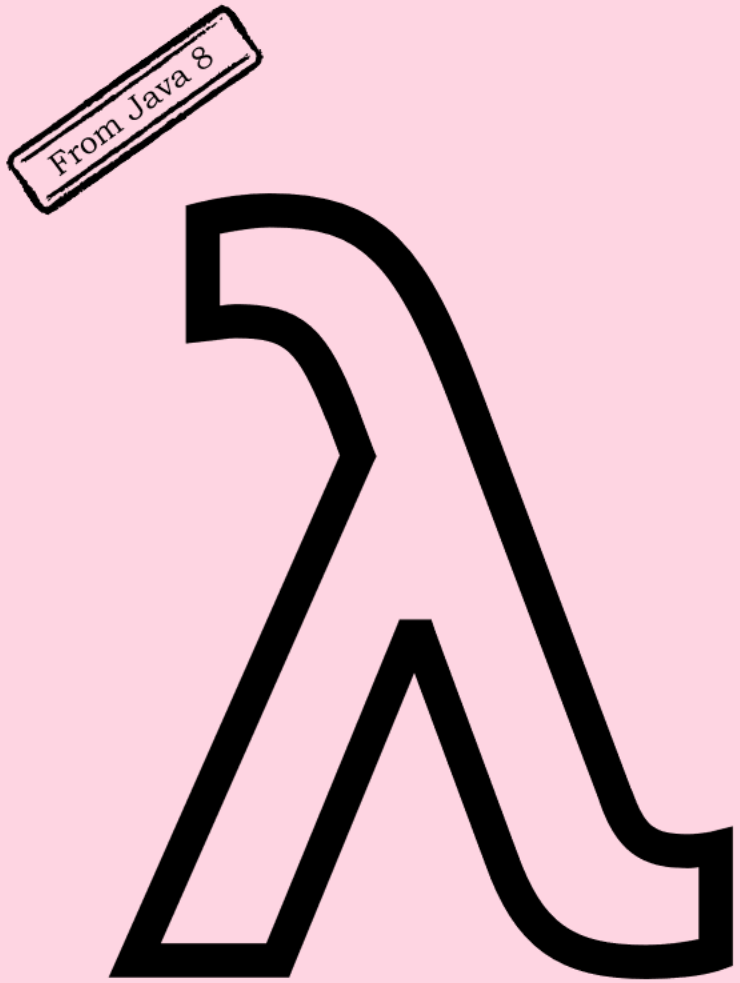
4. Functional programming often employs lazy evaluation, meaning computations are deferred until necessary. This can lead to more efficient use of resources, particularly beneficial when dealing with large datasets.

5. Functional code is more adaptable. It's easier to enhance or alter the logic without worrying about unintended side effects, contributing to a more maintainable and flexible codebase.

6. The functional style allows for code that is well-composed and free from clutter. It reads more like a requirement specification, making it easier to understand and maintain.

What are lambda (λ) expressions ?

eazy bytes



A **lambda expression** is like a compact representation of an anonymous function that can be easily passed around. Unlike a named method, it doesn't have an explicit name, making it simpler to write and comprehend. It shares common characteristics with a function, such as having parameters, a body, a return type, and the potential for a list of exceptions.

That’s a big definition. Let’s break it down,

Anonymous: It lacks a specific name, in contrast to a method, which typically has one. This means there's less to write and think about.

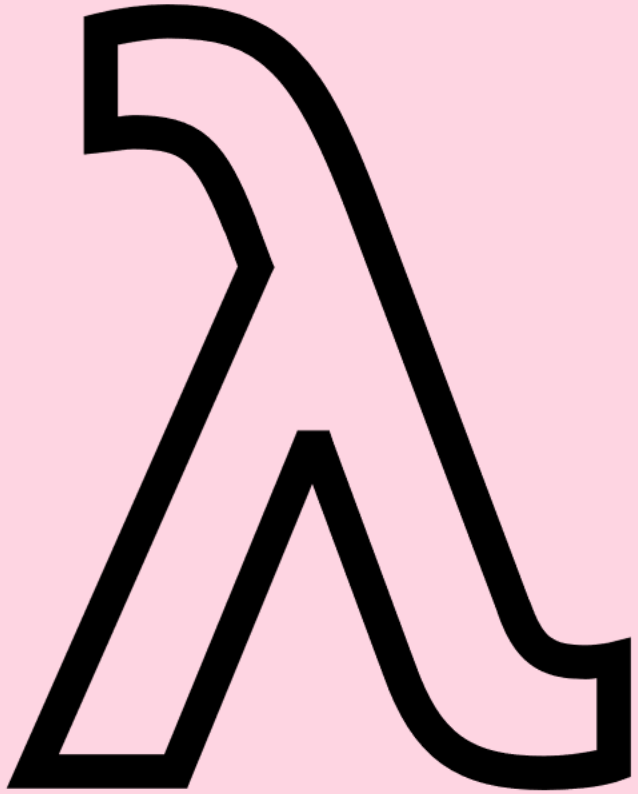
Function: Like a method, a lambda has parameters, a body, a return type, and the option of specifying exceptions. However, it isn't tied to a particular class.

Passed Around: A lambda expression can be conveniently passed as an argument to a method or stored in a variable.

Concise: We don't need to write extensive boilerplate code. Lambda expressions allow for a more streamlined and expressive syntax.

Lambda (λ) expressions syntax

eazy bytes



Below is the syntax for writing lambda expressions that were introduced in Java 8,

Expression lambda:

(parameters-list) -> expression;

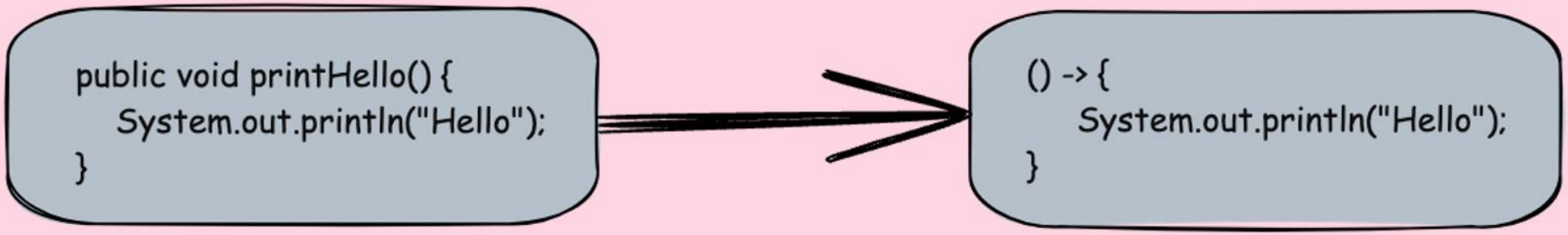
Statement lambda:

(parameters-list) -> { statements; }

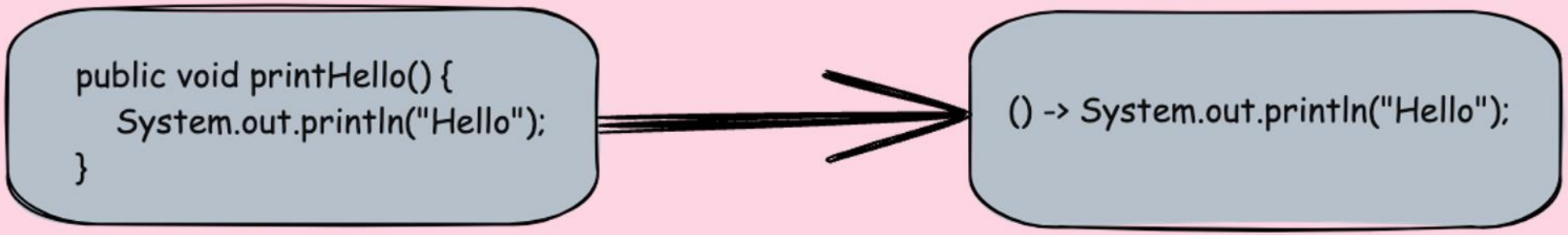
Lambda (λ) expression example1

eazy bytes

Examples of normal method and it's equivalent lambda function,



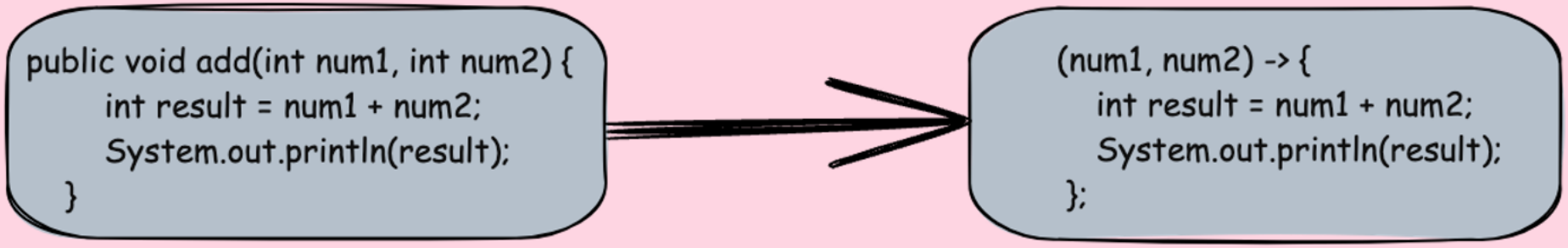
When we have a single line of expression inside a lambda function, we can use the **expression lambda syntax** instead of **statement lambda syntax** to make it more simple.



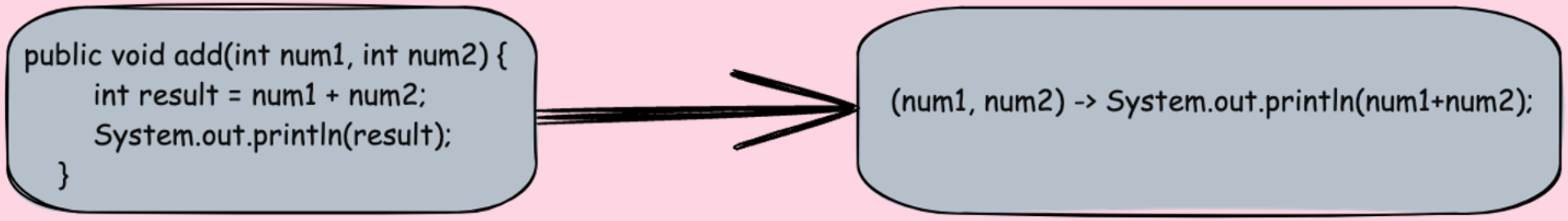
Lambda (λ) expression example2

eazy bytes

Examples of normal method and it's equivalent lambda function,



When we have a single line of expression inside a lambda function, we can use the **expression lambda syntax** instead of **statement lambda syntax** to make it more simple.

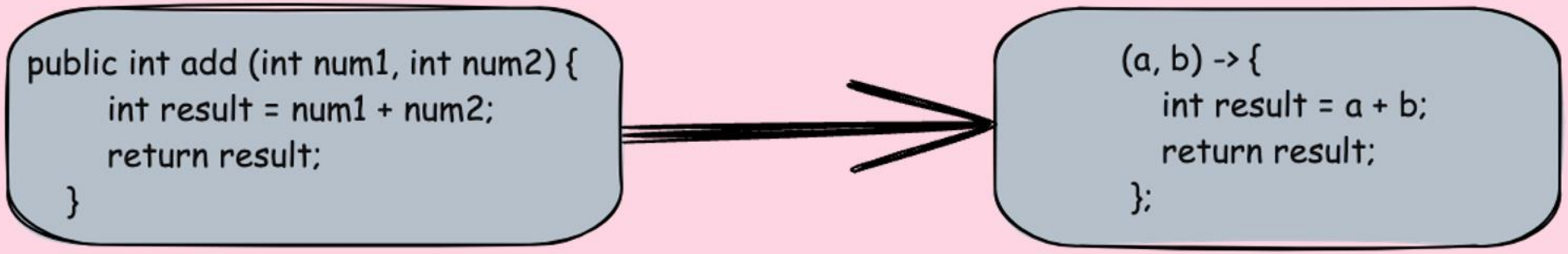


It is not mandatory to use the parameter names same. For example, instead of **num1**, **num2**, we can use **a** and **b** as well inside the lambda function.

Lambda (λ) expression example3

eazy bytes

Examples of normal method and it's equivalent lambda function,



When we have a single line of expression inside a lambda function, we can use the **expression lambda syntax** instead of **statement lambda syntax** to make it more simple.

If needed, you can mention the data type of the lambda function input parameters. For example, instead of **a , b** you can mention **int a , int b**

Functional Interface

eazy bytes

You might be curious about the specific scenarios where lambda expressions are applicable and how the Java compiler works without knowing the method names, parameter data types, and return types ?

We can use a lambda expression in the context of a **functional interface**

A **functional interface** is an interface that specifies exactly a **single abstract method** (SAM). We can use **@FunctionalInterface** annotation as well on top of a interface to make sure that a second abstract method can't be added.

```
public interface MyAdditionInterface {

    int add(int a, int b);

}
```

```
@FunctionalInterface
public interface MySubtractionInterface {

    int subtraction(int a, int b);

}
```

Below are the thumb rules for Functional interfaces,

- Only 1 abstract method is allowed (Also called as SAM Single Abstract method)
- Any number of default methods are allowed
- Any number of static methods are allowed
- Any number of private methods are allowed

Functional Interface

eazy bytes

Which of the following interfaces are functional interfaces ?

```
public interface Hello {
    void sayHello();
}
```

Hello is a functional interface as it contains only a single abstract method

```
public interface Print extends Hello {
    String print(String msg);
}
```

Print isn't a functional interface as it has two abstract methods called print and sayHello (inherited from Hello).

```
public interface Empty {

}
```

Empty isn't a functional interface because it declares no abstract method.

Functional Interface

eazy bytes

Which of the following interfaces are functional interfaces ?

```
@FunctionalInterface
public interface ArithmeticOperation {

    public int performOperation (int a, int b);
    public int performOperation (int c);

}
```

Invalid functional interface as it contains two abstract methods

```
@FunctionalInterface
public interface A {
    public String m1 (String input);
}


@FunctionalInterface
public interface B extends A {
    public String m1 (String input);
}
```

interface B is a valid functional interface though it inherits 1 abstract method from its parent and it also has 1. This is because the abstract method name and signature is same as parent Interface A

```
public interface C {

    public void m2 (int num);

}
```

Compiler will not have any problem with interface C though it is not marked as a Functional interface. As long as it have a single abstract method, it can be used for lambda expression

How Functional Interface & Lambda expressions are linked ?

eazy bytes

Functional interfaces in Java allow you to use lambda expressions to provide the implementation of their single abstract method directly within your code. With lambda expressions, you can define and use these interfaces more conveniently, treating the entire expression as an instance of the functional interface. In simpler terms, functional interfaces and lambda expressions make it easier to write and use concise, inline implementations of specific functionalities in your Java code.

Let's consider a functional interface and try to understand how to use lambda expressions using the same,

```
public interface Hello {
    void sayHello();
}
```

Imagine you have a functional interface **Hello** with a single abstract method **sayHello()** inside it

```
public static void process(Hello h) {
    h.sayHello();
}
```

Think like, we have a static method **process()** in some class and this method accepts a input parameter 'h' of type **Hello**. Using the same parameter variable, the method **sayHello()** is going to be invoked like shown here.

How Functional Interface & Lambda expressions are linked ?

eazy bytes

```
public interface Hello {
    void sayHello();
}
```

```
public static void process(Hello h) {
    h.sayHello();
}
```

Approach 1

```
Hello h1 = () -> System.out.println("Hello");
process(h1);
```

Approach 2

```
process(() -> System.out.println("Hi"));
```

The signature of the abstract method of the functional interface describes the signature of the lambda expression.Compiler can infer the details like method name, data type of parameters, return type from the abstract method. That's why the abstract method inside functional interface is also called as a **function descriptor**

We can pass the lambda expression as an input either using a variable of type belongs to functional interface or directly. With lambda expression, we are passing the behaviour or business logic as an input. Just like how different objects holds different data, with lambda expressions we can pass different business logics as input. We don't have to settle with a single logic like in imperitive approach.

Anonymous inner class in the place of lambda expressions

eazy bytes

In the place of lambda expressions, we can use Anonymous inner class as well to provide the behaviour as input but it looks clumsier and complex like shown below. That's why it is not recommended to use this approach though it is available from day 1 of Java,

```
public interface Hello {
    void sayHello();
}
```

```
public static void process(Hello h) {
    h.sayHello();
}
```

Anonymous inner class approach

```
Hello h2 = new Hello() {
    public void sayHello() {
        System.out.println("Anonymous");
    }
};

process(h2);
```

Anonymous inner class Vs Lambda expression

eazy bytes

Anonymous inner class

It is a anonymous inner class with out name

It can implement the interfaces with any number of abstract methods inside them

They can be instantiated and can extend abstract and concrete classes.

Instance variables can be declared and “this” inside it always refer to the current inner class

Memory inside heap will be allocated on demand whenever we create an object for it

Lambda expression

It is a anonymous method with out name

It can implement the interfaces which has only 1 abstract method called Functional interfaces

They can’t be instantiated and can’t extend abstract and concrete classes.

Only local variables can be declared and “this” inside it always refer to the outer enclosing class

Permanent memory (Method area) will be allocated for it

Functional interfaces inside JDK

eazy bytes

To make developer life easy around developing lambda expressions, Java 8 has provided some pre defined functional interfaces by considering most common requirements during the development. All such interfaces are present inside the **java.util.function** package

Below are the most important functional interfaces provided by the Java team,

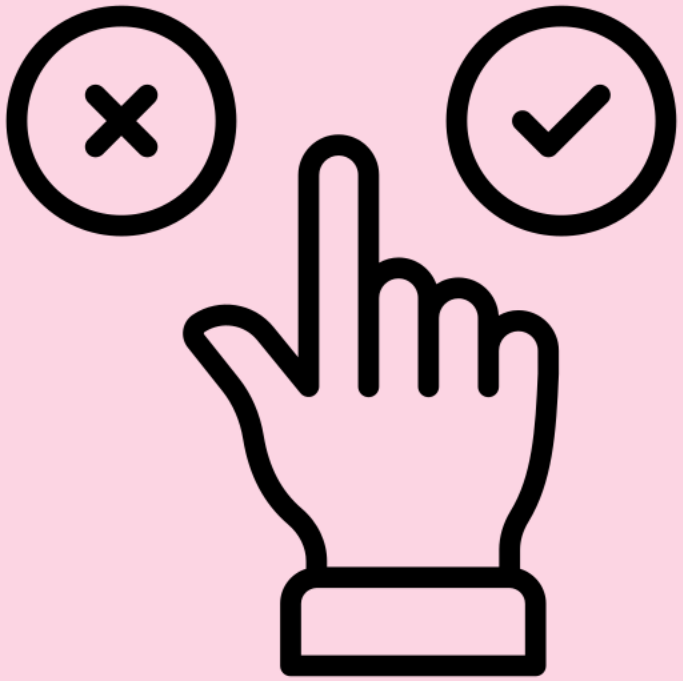
- java.util.function.Predicate <T>
- java.util.function.Function <T,R>
- java.util.function.Consumer <T>
- java.util.function.Supplier <T>
- java.util.function.BiPredicate <T,U>
- java.util.function.BiFunction <T,U,R>
- java.util.function.BiConsumer <T,U>
- java.util.function. UnaryOperator <T>
- java.util.function. BinaryOperator <T>
- Primitive Functional Interfaces

java.util.function.Predicate <T>

eazy bytes

Predicate Functional interface handles the scenarios where we accept a input parameter and return the boolean after processing the input

@param <T> - the type of the input to the function



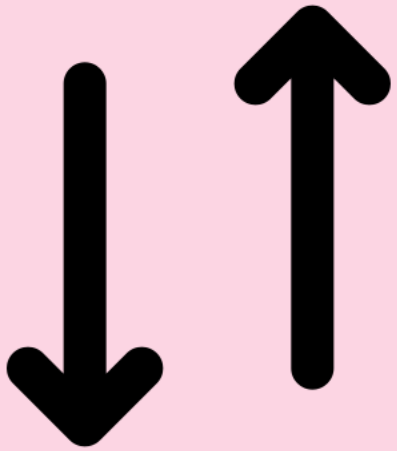
- boolean test(T t); ===> Single abstract method available
- default Predicate<T> and(Predicate<? super T> other) ===> Default method that can be used while joining multiple predicate conditions. This acts like a logical AND condition
- default Predicate<T> negate() ===> Default method that can be used while joining multiple predicate conditions. This acts like a logical NOT condition
- default Predicate<T> or(Predicate<? super T> other) ===> Default method that can be used while joining multiple predicate conditions. This acts like a logical OR condition

java.util.function.Function<T,R>

eazy bytes

Function is similar to Predicate except with a change that instead of boolean it can return any datatype as outcome. It represents a function that accepts one argument and produces a result

@param <T> - the type of the input to the function
@param <R> - the type of the result from the function



- R apply(T t); ===> Single abstract method available
- static <T> Function<T, T> identity() ===> Utility static method which will return the same input value as output
- default <V> Function<V, R> compose(Function<? super V, ? extends T> before) ===> Default method that can be used for chaining
- default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) ===> Default method that can be used for chaining

The difference between andThen() and compose() is that in the andThen first func will be executed followed by second func whereas in compose it is vice versa

Predicate Vs Function

eazy bytes

Predicate

Is used for checking the conditions on given input and return a boolean value

The single abstract method(SAM) name is test()

It takes one type parameter which indicates Input parameter and return parameter is always boolean

It has a static method called isEqual() which will check the equality of the 2 values/objects.

It has 3 default methods for chaining namely and(), or() & negate()

Function

Is used for executing business logic for the input and return any type of output.

The single abstract method(SAM) name is apply()

It takes 2 type of parameters which indicates input parameter and return parameter

It has a static method called identity() which will return the same as input given.

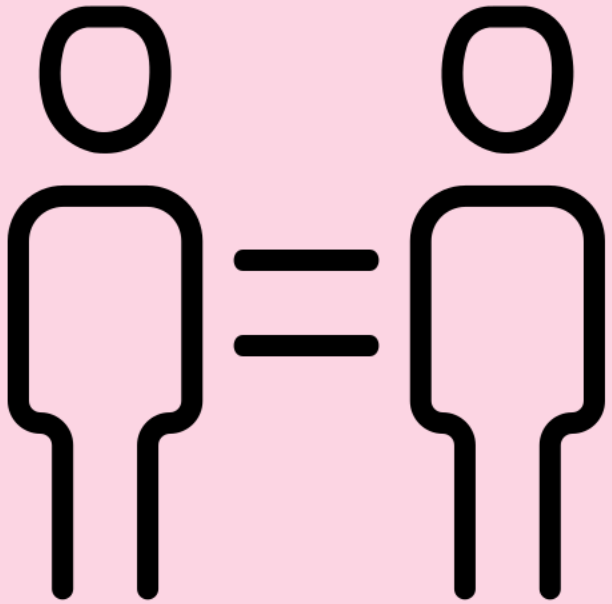
It has 2 default methods for chaining namely andThen() & compose()

java.util.function.UnaryOperator<T>

eazy bytes

If we have scenarios where both the input and output parameters data type is same, then instead of using Function<T,R> we can use the UnaryOperator <T>

@param <T> - the type of the operand and result of the operator



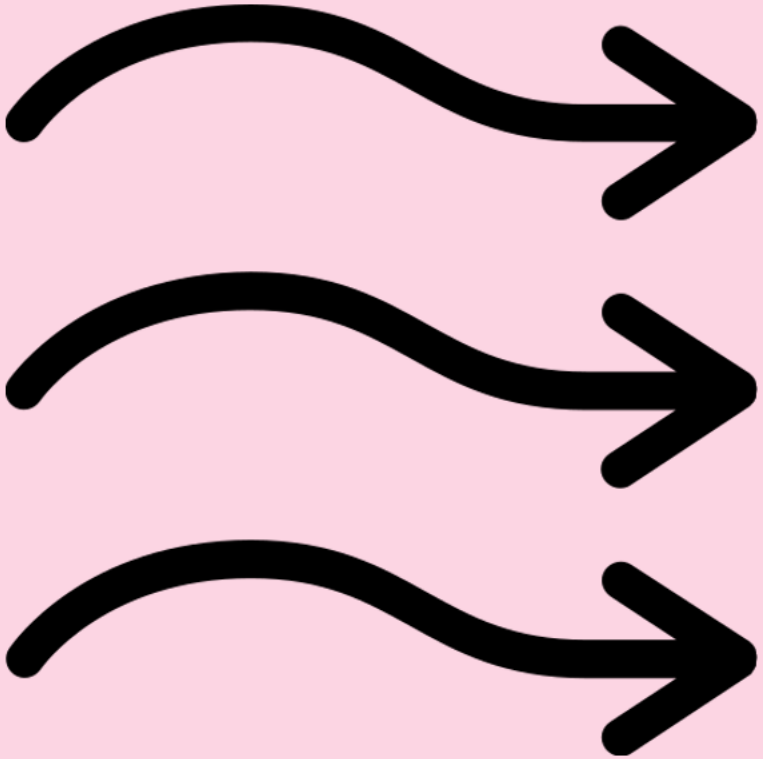
- It is a child of **Function<T,T>** . So all the methods apply(), compose(), andThen() are available inside the **UnaryOperator** interface also.
- In other words we can say that **UnaryOperator** takes one argument, and returns a result of the same type of its arguments

java.util.function.Consumer<T>

eazy bytes

As the name indicates Consumer interface will always consumes/accept the given input for processing but not return anything to the invocation method

@param <T> - the type of the input to the function



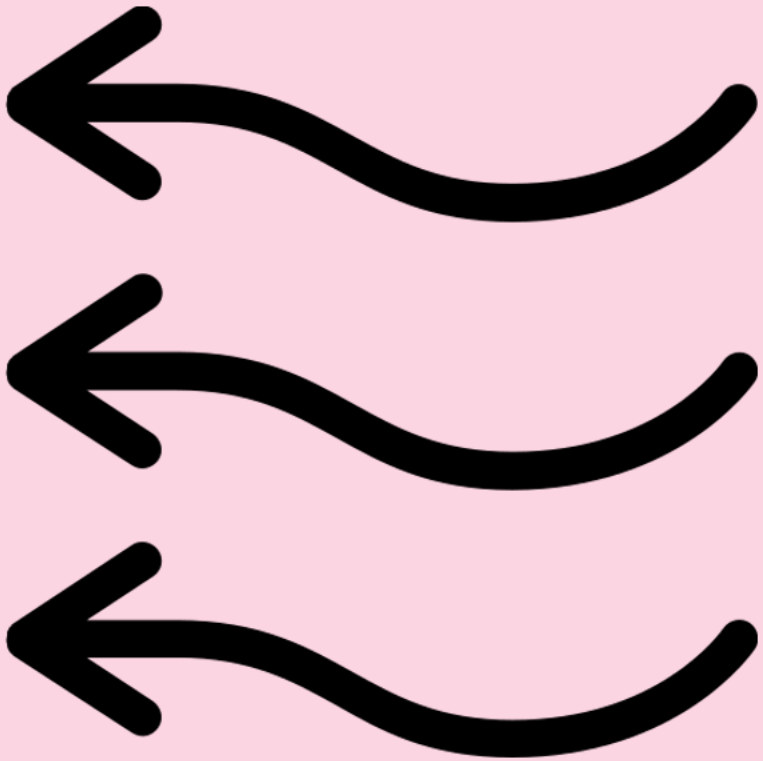
- void accept(T t); ===> Single abstract method available
- default Consumer<T> andThen(Consumer<? super T> after)
  ===> Default method that can be used for chaining
- No static methods are available in Consumer functional interface

java.util.function.Supplier<T>

eazy bytes

As the name indicates Supplier interface will always return a value with out accepting any input. Think of the scenarios like generating report or OTP where we don't provide any input.

@param <T> - the type of result supplied by this supplier

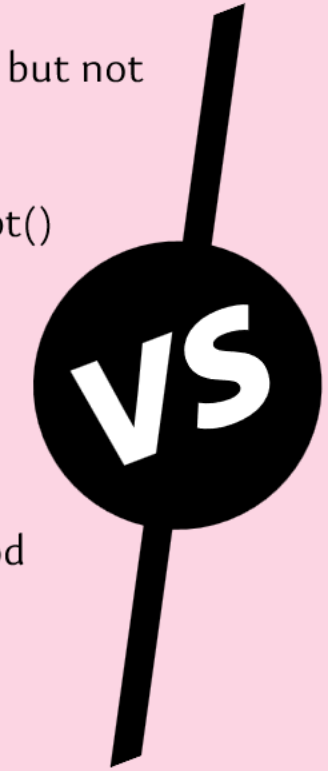


- T get(); ===> Single abstract method available
- There are no static and chaining methods available in Supplier functional interface. The reason is that it will not accept any input so there is no meaning of chaining in it

Consumer Vs Supplier

eazy bytes

Consumer	Supplier
Is used in the scenarios where we send an input but not expect any return value from it	Is used in the scenarios where we don't send any input but expecting return value from it
The single abstract method(SAM) name is accept()	The single abstract method(SAM) name is get()
It takes one type parameter which indicates Input parameter and return parameter is always void	It takes one type parameter which indicates output parameter and input parameter is not needed
It has no static methods but has 1 default method andThen() for chaining	It has no static and default methods inside it
It is like setter method inside our POJO classes	It is like getter method inside our POJO classes



BiFunctional Interfaces

eazy bytes

- ✓ As of now we saw the functional interfaces which accepts only 1 parameter as input but what if we have a need to send 2 input parameters. To address the same Java has **BiFunctional interfaces**

- ✓ **java.util.function.BiPredicate<T, U>** - Similar to Predicate but it can accept 2 input parameters and return a boolean value

  - @param <T> - the type of the first argument to the BiPredicate
  - @param <U> - the type of the second argument the BiPredicate

- ✓ **java.util.function.BiFunction<T, U, R>** - Similar to Function but it can accept 2 input parameters and return a output as per the data type mentioned.

  - @param <T> - the type of the first argument to the BiFunction
  - @param <U> - the type of the second argument the BiFunction
  - @param <R> - the type of the return type from the BiFunction

BiFunctional Interfaces

eazy bytes

- ✓ **java.util.function.BiConsumer<T, U>** - Similar to Consumer but it can accept 2 input parameters and no return value same as Consumer

  - @param <T> - the type of the first argument to the BiConsumer
  - @param <U> - the type of the second argument the BiConsumer

- ✓ There is no **BiSupplier** corresponding to Supplier functional interface as it will not accept any input parameters

- ✓ **java.util.function.BinaryOperator<T>** - It is a child of BiFunction<T,U,R> .
  We will use this in the scenarios where the 2 input parameters and 1 return parameter data types is same.

  - @param <T> the type of the input parameters and return type

  In addition to the methods that it inherits from BiFunction<T,U,R>, it also has 2 utility static methods inside it.
  They both will be used to identify the minimum or maximum of 2 elements based on the comparator logic that we pass,

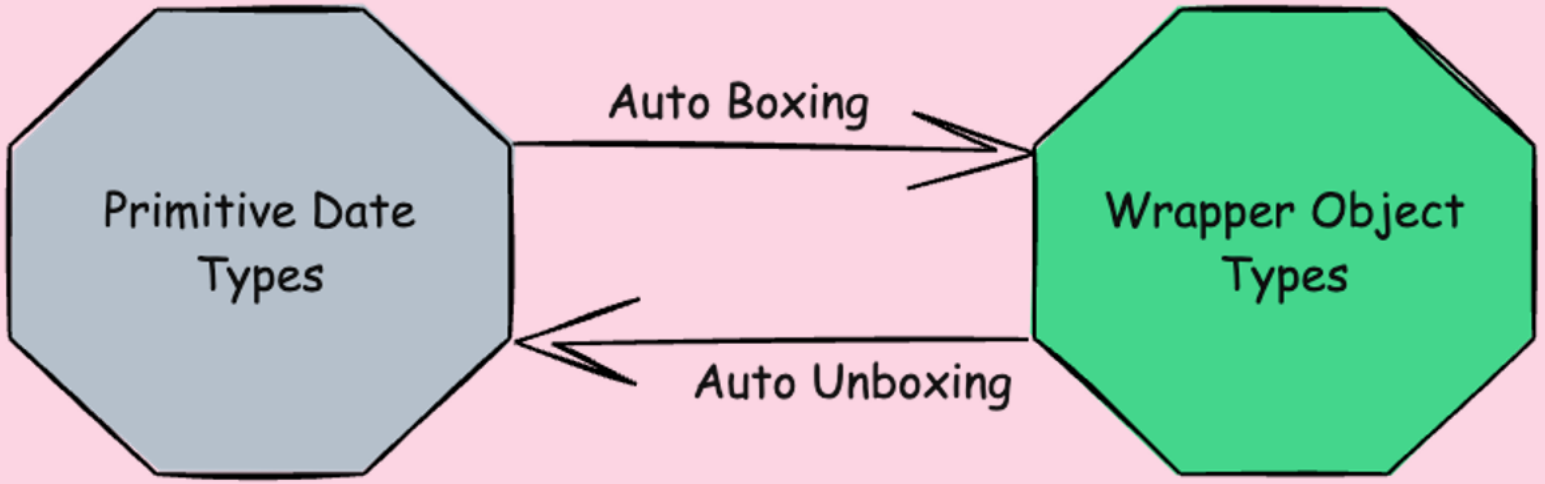
  - static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
  - static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)

Primitive Type functional interfaces

eazy bytes

- ✓ All the functional interfaces like Predicate, Function, Consumer that we discussed previously accepts or returns only Object type values like Integer, Double, Float, Long etc.

- ✓ There is a performance problem with this. If we pass any primitive input values like int, long, double, float Java will auto box them in order to convert them into corresponding wrapper objects. Similarly once the logic is being executed Java has to convert them into primitive types using unboxing.



- ✓ Since there is lot of auto boxing and unboxing happening it may impact performance for larger values/inputs. To overcome such scenarios Java has primitive type functional interfaces as well

Primitive Type functional interfaces

eazy bytes

Below are the most important primitive functional interfaces provided by the Java team,

- ✓ **java.util.function.IntPredicate –** Always accepts int as input

```
boolean test(int value);
```

- ✓ **java.util.function.DoublePredicate –** Always accepts double as input

```
boolean test(double value);
```

- ✓ **java.util.function.LongPredicate –** Always accepts long as input

```
boolean test(long value);
```

- ✓ **java.util.function.IntFunction<R> –** Always accepts int as input and return any type as output

```
R apply(int value);
```

Primitive Type functional interfaces

eazy bytes

Below are the most important primitive functional interfaces provided by the Java team,

- ✓ **java.util.function.DoubleFunction<R>** – Always accepts double as input and return any type as output

```
R apply(double value);
```

- ✓ **java.util.function.LongFunction<R>** – Always accepts long as input and return any type as output

```
R apply(long value);
```

- ✓ **java.util.function.ToIntFunction<T>** – Always return int value but accepts any type as input

```
int applyAsInt(T value);
```

- ✓ **java.util.function.ToDoubleFunction<T>** – Always return double value but accepts any type as input

```
double applyAsDouble(T value);
```

Primitive Type functional interfaces

eazy bytes

Below are the most important primitive functional interfaces provided by the Java team,

- ✓ **java.util.function.ToLongFunction<T> –** Always return long value but accepts any type as input

```
long applyAsLong(T value);
```

- ✓ **java.util.function.IntToLongFunction –** Always takes int type as input and return long as return type

```
long applyAsLong(int value);
```

- ✓ **java.util.function.IntToDoubleFunction –** Always takes int type as input and return double as return type

```
double applyAsDouble(int value);
```

- ✓ **java.util.function.LongToIntFunction –** Always takes long type as input and return int as return type

```
int applyAsInt(long value);
```

Primitive Type functional interfaces

eazy bytes

Below are the most important primitive functional interfaces provided by the Java team,

- ✓ **java.util.function.LongToDoubleFunction –** Always takes long type as input and return double as return type

```
double applyAsDouble(long value);
```

- ✓ **java.util.function.DoubleToIntFunction –** Always takes double as input and return int as return type

```
int applyAsInt(double value);
```

- ✓ **java.util.function.DoubleToLongFunction –** Always takes double type as input and return long as return type

```
long applyAsLong(double value);
```

- ✓ **java.util.function.ToIntBiFunction –** Always takes 2 input parameters of any type and return int as return type

```
int applyAsInt(T t, U u);
```

Primitive Type functional interfaces

eazy bytes

Below are the most important primitive functional interfaces provided by the Java team,

- ✓ **java.util.function.ToLongBiFunction –** Always takes 2 input parameters of any type and return long as return type

```
long applyAsLong(T t, U u);
```

- ✓ **java.util.function.ToDoubleBiFunction –** Always takes 2 input parameters of any type and return double as return type

```
double applyAsDouble(T t, U u);
```

- ✓ **java.util.function.IntConsumer –** Always accepts int value as input

```
void accept(int value);
```

- ✓ **java.util.function.LongConsumer –** Always accepts long value as input

```
void accept(long value);
```

Primitive Type functional interfaces

eazy bytes

Below are the most important primitive functional interfaces provided by the Java team,

- ✓ **java.util.function.DoubleConsumer –** Always accepts double value as input

```
void accept(double value);
```

- ✓ **java.util.function.ObjIntConsumer<T> –** Always accepts 2 inputs of type int and any data type

```
void accept(T t, int value);
```

- ✓ **java.util.function.ObjLongConsumer<T> –** Always accepts 2 inputs of type long and any data type

```
void accept(T t, long value);
```

- ✓ **java.util.function.ObjDoubleConsumer<T> –** Always accepts 2 inputs of type double and any data type

```
void accept(T t, double value);
```

Primitive Type functional interfaces

eazy bytes

Below are the most important primitive functional interfaces provided by the Java team,

- ✓ **java.util.function.IntSupplier –** Always return int value as output

```
int getAsInt();
```

- ✓ **java.util.function.LongSupplier –** Always return long value as output

```
long getAsLong();
```

- ✓ **java.util.function.DoubleSupplier –** Always return double value as output

```
double getAsDouble();
```

- ✓ **java.util.function.BooleanSupplier –** Always return boolean value as output

```
boolean getAsBoolean();
```

Primitive Type functional interfaces

eazy bytes

Below are the most important primitive functional interfaces provided by the Java team,

- ✓ **java.util.function.IntUnaryOperator –** Always accept and return int values

```
int applyAsInt(int operand);
```

- ✓ **java.util.function.LongUnaryOperator –** Always accept and return long values

```
long applyAsLong(long operand);
```

- ✓ **java.util.function.DoubleUnaryOperator –** Always accept and return double values

```
double applyAsDouble(double operand);
```

Primitive Type functional interfaces

eazy bytes

Below are the most important primitive functional interfaces provided by the Java team,

- ✓ **java.util.function.IntBinaryOperator –** Always accept 2 int input parameters and return an int value

```
int applyAsInt(int left, int right);
```

- ✓ **java.util.function.LongBinaryOperator –** Always accept 2 long input parameters and return long value

```
long applyAsLong(long left, long right);
```

- ✓ **java.util.function.DoubleBinaryOperator –** Always accept 2 double input parameters and return double value

```
double applyAsDouble(double left, double right);
```

Lexical Scoping

eazy bytes

Despite the visual resemblance of a lambda expression to a method declaration, it does not establish an independent scope. Instead, it operates within the scope of its enclosing context. This characteristic is referred to as lexical scoping for lambda expressions. For instance, when a lambda expression is utilized within a method, it remains within the scope of that method.

✓ The below code of a lambda expression inside the main() method generates a compile-time error, as its parameter name input is already defined in the main() method's scope:

```
@FunctionalInterface
public interface Printer {
   void print(String input);
}

public static void main(String[] args) {
    String input = "Hello World";
    Printer printer = input -> System.out.println(input); // Compile error
}
```

✓ The below code generates a compile-time error for the same reason that the local variable named input is in scope inside the body of the lambda expression, and the lambda expression is attempting to reinitialize/declare a local variable with the same name input:

Any local variable accessed inside a lambda expression must be effectively final. The lambda expression attempts to modify the input variable inside its body, and that causes the compile-time error.

```
public static void main(String[] args) {
    String input = "Hello World";
    Printer printer = input1 -> {
            input = "Hi World"; // Compilation error
            System.out.println(input1);
        };
    }
}
```

this and super inside lambda expression

eazy bytes

The meanings of the keywords 'this' and 'super' remain consistent inside a lambda expression and its enclosing method. It's important to note that this behavior differs from the meanings of these keywords within a local and anonymous inner class. In the context of such inner classes, the keyword 'this' refers to the current instance of the inner class itself, not its enclosing class.

```java
public class ThisDemo {

    public static void main(String[] args) {
        ThisDemo thisDemo = new ThisDemo();
        Printer lambdaPrinter = thisDemo.getLambdaPrinter();
        lambdaPrinter.print("Lambda Expression");
        Printer anonymousPrinter = thisDemo.getAnonymousPrinter();
        anonymousPrinter.print("Anonymous Inner Class");
    }

    public Printer getLambdaPrinter() {
        Printer printer = msg -> System.out.println(msg + " : "+ this.getClass());
        return printer;
    }

    public Printer getAnonymousPrinter() {
        Printer printer = new Printer() {
            @Override
            public void print(String input) {
                System.out.println(input + " : "+ this.getClass());
            }
        };
        return printer;
    }

}
```

```
Lambda Expression : class com.eazybytes.lambda.ThisDemo
Anonymous Inner Class : class com.eazybytes.lambda.ThisDemo$1
```