
Java Lambda Expression
&
Streams API
Bok, Jong Soon
javaexpert@nate.com
https://github.com/swacademy/Core-Java

What's Functional Programming?

- Is a programming paradigm
  - Treats computation as the evaluation of mathematical functions
  - Avoids changing state and mutable data.
- Is about building software by
  - Composing pure functions
  - Avoiding shared state
  - Emphasizing immutability and declarative constructs.
- Provides a different approach compared to traditional imperative and object-oriented programming, often resulting in more predictable and maintainable code.

Key Concepts & Characteristics of Functional Programming

- First-Class and Higher-Order Functions
  - First-Class Functions
    - Functions are treated as first-class citizens, meaning they can be assigned to variables, passed as arguments, and returned from other functions.
  - Higher-Order Functions
    - Functions that can take other functions as arguments or return them as results.
- Pure Functions
  - Is a function where the output value is determined only by its input values, without observable side effects.
  - This means given the same input, a pure function will always return the same output.
- Immutability
  - Data is immutable, meaning once it is created, it cannot be changed.
  - Instead of modifying existing data, new data structures are created.

Key Concepts & Characteristics of Functional Programming (Cont.)

- Recursion
  - Functional programming often uses recursion instead of loops for iteration.
  - Recursive functions call themselves with modified parameters until a base condition is met.
- Referential Transparency
  - An expression is referentially transparent if it can be replaced with its value without changing the program's behavior.
  - This property makes reasoning about and testing code easier.
- Lazy Evaluation
  - Delaying the evaluation of an expression until its value is actually needed.
  - This can improve performance by avoiding unnecessary calculations.

Examples of Functional Programming Languages

- Haskell
- Lisp
- Scala
- Erlang
- F#
- JavaScript(ECMAScript)
- Java(Java Version 8 Higher)

Benefits of Functional Programming

- Modularity
  - Encourages writing small, reusable functions.
- Predictability
  - Pure functions and immutability make the code more predictable and easier to debug.
- Concurrency
  - Easier to write concurrent and parallel programs since immutable data structures eliminate race conditions.

Use Cases of Functional Programming

- Data Analysis
  - For handling transformations and data pipelines.
- Concurrent and Parallel Programming
  - For building systems that need to handle a lot of simultaneous tasks.
- Reactive Programming
  - For developing applications that respond to changes in data over time.

Lambda Expressions

- Think about :

```
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("button clicked");
    }
});
```

```
button.addActionListener(event -> System.out.println("button clicked"));
```

Lambda Expressions (Cont.)

- Some different ways of writing lambda expression:

```
Runnable noArguments = () -> System.out.println("Hello World");  ❶

ActionListener oneArgument = event -> System.out.println("button clicked");  ❷

Runnable multiStatement = () -> {  ❸
   System.out.print("Hello");
   System.out.println(" World");
};

BinaryOperator<Long> add = (x, y) -> x + y;  ❹

BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;  ❺
```

What's Functional Interface in Java?

- Is an interface that contains exactly one abstract method.
- These interfaces can have multiple default or static methods but must have only one abstract method.
- Provide target types for lambda expressions and method references, enabling functional programming in Java.

What's Functional Interface in Java? (Cont.)

- Single Abstract Method
  - It has exactly one abstract method.
  - This method defines the operation that can be performed by the interface.
- `@FunctionalInterface` Annotation
  - Is not mandatory but is highly recommended.
  - It helps to enforce the rule that the interface can only have one abstract method.
  - If an interface annotated with `@FunctionalInterface` has more than one abstract method, the compiler will generate an error.

What's Functional Interface in Java? (Cont.)

- Default and Static Methods
  - An have multiple default or static methods.
  - These methods do not count as abstract methods and can provide additional functionality or utility methods.
- Compatibility with Lambda Expressions
  - Are designed to be used with lambda expressions, method references, and constructor references, which allows for concise and readable functional-style code.

Common Functional Interfaces in Java

- The `java.util.functioin` package contains several pre-defined functional interfaces which are common used:

Interface name	Arguments	Returns	Example
`Predicate<T>`	`T`	`boolean`	Has this album been released yet?
`Consumer<T>`	`T`	`void`	Printing out a value
`Function<T,R>`	`T`	`R`	Get the name from an `Artist` object
`Supplier<T>`	None	`T`	A factory method
`UnaryOperator<T>`	`T`	`T`	Logical not (`!`)
`BinaryOperator<T>`	`(T, T)`	`T`	Multiplying two numbers (`*`)

Common Functional Interfaces in Java (Cont.)

- **`Function<T,R>`**:
  - Represents a function that takes an argument of type `T` and produces a result of type `R`.

```
Function<String, Integer> lengthFunction = (String s) -> s.length();
```

- **`Consumer<T>`**:
  - Represents an operation that takes a single input argument and returns no result.

```
Consumer<String> printConsumer = (String s) -> System.out.println(s);
```

- **`Supplier<T>`**:
  - Represents a supplier of results.
  - It does not take any arguments but returns a result.

```
Supplier<Double> randomSupplier = () -> Math.random();
```

Common Functional Interfaces in Java (Cont.)

- **`Predicate<T>`**:
  - Represents a predicate (boolean-valued function) of one argument.

```
Predicate<Integer> isEvenPredicate = (Integer i) -> i % 2 == 0;
```

- **`UnaryOperator<T>`**:
  - Represents an operation on a single operand that produces a result of the same type as its operand.

```
UnaryOperator<Integer> squareOperator = (Integer x) -> x * x;
```

- **`BinaryOperator<T>`**:
  - Represents an operation upon two operands of the same type, producing a result of the same type as the operands.

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

Example of Custom Functional Interface

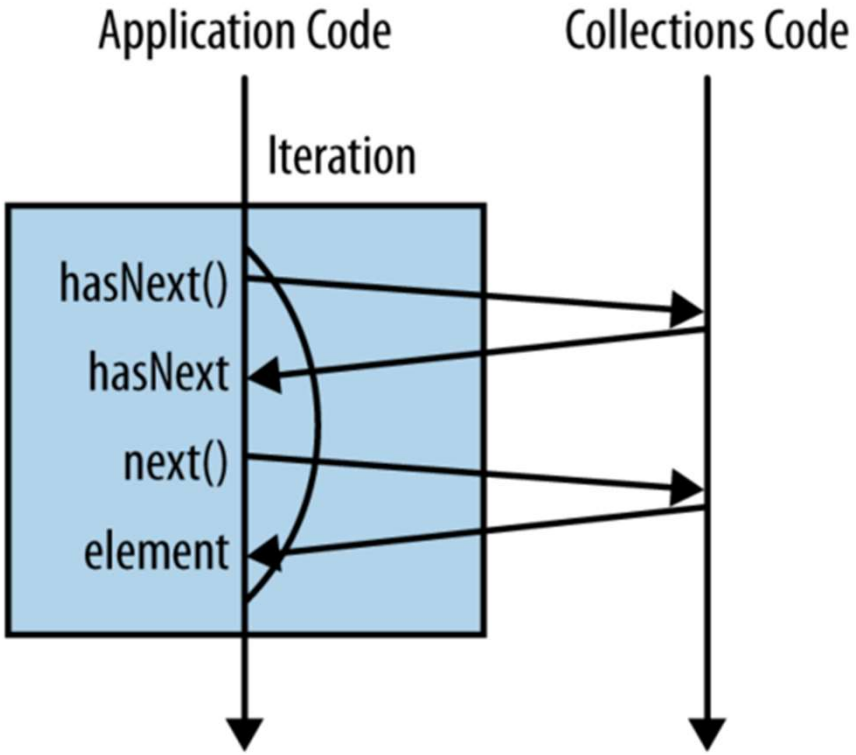
```java
@FunctionalInterface
interface MyFunctionalInterface {
    void execute();

    // You can have other non-abstract methods (default or static)
    default void defaultMethod() {
        System.out.println("This is a default method");
    }

    static void staticMethod() {
        System.out.println("This is a static method");
    }
}
```

```java
public class FunctionalInterfaceExample {
    public static void main(String[] args) {
        // Using lambda expression to implement the execute method
        MyFunctionalInterface myFunction = () -> System.out.println("Executing...");

        // Calling the method
        myFunction.execute();

        // Calling the default method
        myFunction.defaultMethod();

        // Calling the static method
        MyFunctionalInterface.staticMethod();
    }
}
```

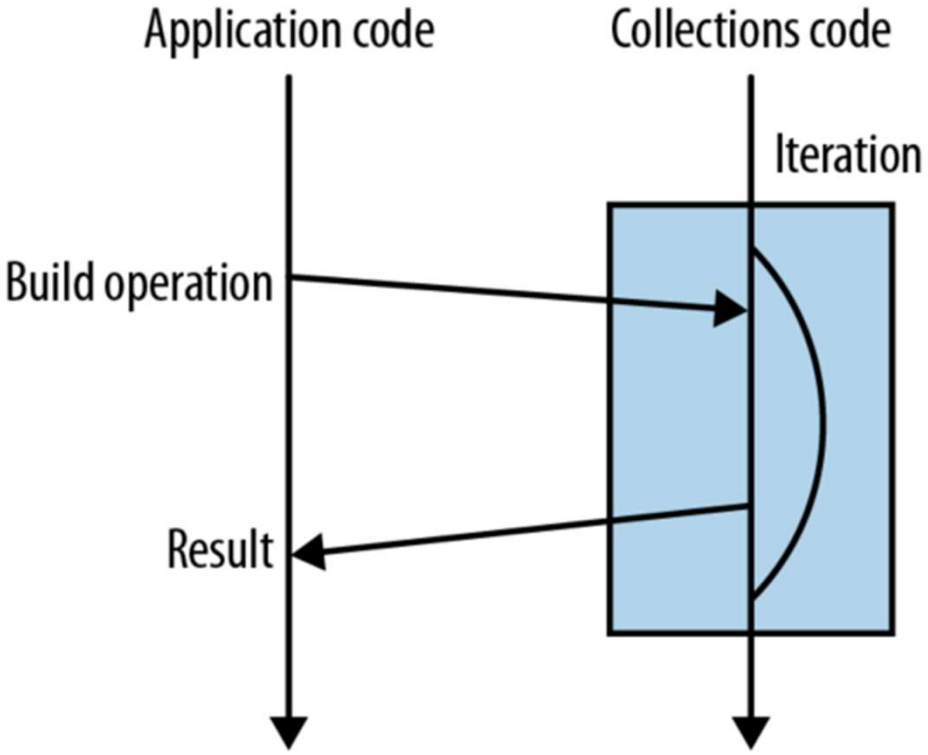
Streams API

- Introduced in Java 8.
- Is a powerful tool for processing sequences of elements in a functional-style manner.
- Allows developers to perform operations on collections of data in a concise and efficient way.
- Provides a powerful and flexible way to process collections of data, enabling developers to write clean, efficient, and expressive code using functional programming techniques.

Streams API (Cont.)



External iteration



Internal iteration

Streams API's Key Concepts

- Stream Definition:
  - Is a sequence of elements supporting sequential and parallel aggregate operations.
  - Unlike collections, streams are not data structures that store elements.
  - Instead, streams convey elements from a source (such as a collection) through a pipeline of computational operations.

Streams API's Key Concepts (Cont.)

- Pipeline of Operations:
  - Streams are processed through a pipeline of operations.
  - The pipeline consists of:
    - Source: The data source (e.g., collection, array, I/O channel).
    - Intermediate Operations:
      - Operations that transform the stream, producing another stream (e.g., `filter`, `map`, `sorted`).
      - These operations are lazy and do not execute until a terminal operation is called.
    - Terminal Operations:
      - Operations that produce a result or a side-effect (e.g., `forEach`, `collect`, `reduce`).
      - Terminal operations trigger the execution of the entire stream pipeline.
- Functional Interfaces:

Streams API's Key Concepts (Cont.)

- Functional Interfaces:
  - The Streams API relies heavily on functional interfaces such as `Function`, `Consumer`, `Predicate`, and `Supplier` to pass behavior as parameters to stream operations..

Streams API Common Operations

- Creation
  - Streams can be created from various data sources such as collections, arrays, or generators.

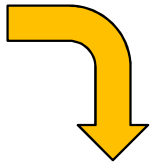
```
1   List<String> list = Arrays.asList("a", "b", "c");
2   Stream<String> stream = list.stream();
3
4   Stream<Integer> intStream = Stream.of(1, 2, 3, 4);
5
6   Stream<Double> randomStream = Stream.generate(Math::random);
```

Streams API Common Operations (Cont.)

- Intermediate Operations
  - **`filter`**: Filters elements based on a predicate.

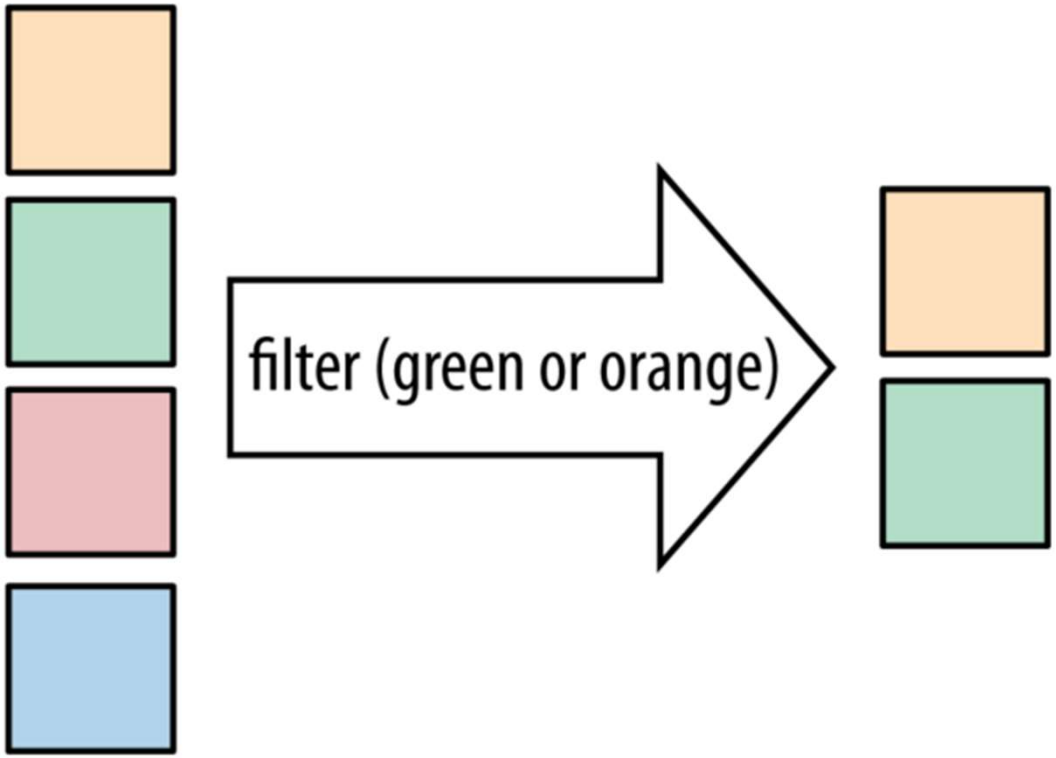
```
1   List<String> filtered = list.stream()
2                            .filter(s -> s.startsWith("a"))
3                            .collect(Collectors.toList());
```

```
List<String> beginningWithNumbers = new ArrayList<>();
for(String value : asList("a", "1abc", "abc1")) {
    if (isDigit(value.charAt(0))) {
        beginningWithNumbers.add(value);
    }
}

assertEquals(asList("1abc"), beginningWithNumbers);
```



```
List<String> beginningWithNumbers
  = Stream.of("a", "1abc", "abc1")
          .filter(value -> isDigit(value.charAt(0)))
          .collect(toList());

assertEquals(asList("1abc"), beginningWithNumbers);
```

Streams API Common Operations (Cont.)



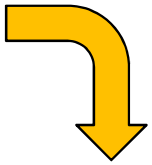
The `filter` operation

Streams API Common Operations (Cont.)

- Intermediate Operations
  - **`map`**: Transforms each element using a function.

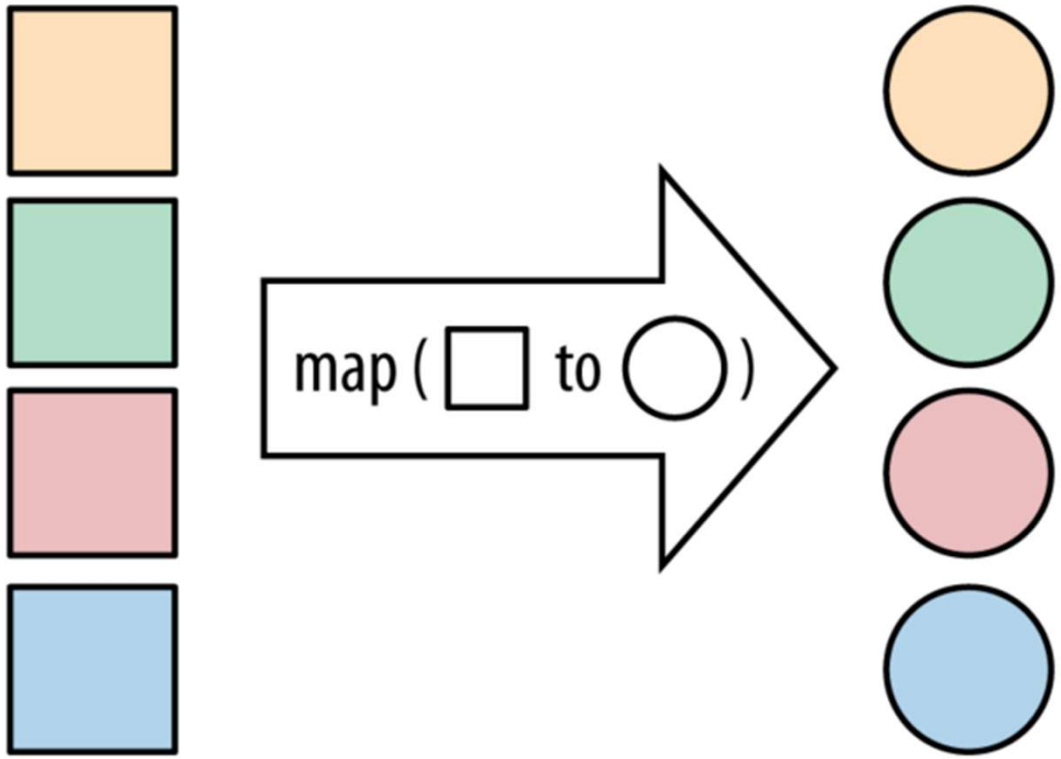
```
1   List<Integer> lengths = list.stream()
2                            .map(String::length)
3                            .collect(Collectors.toList());
```

```
List<String> collected = new ArrayList<>();
for (String string : asList("a", "b", "hello")) {
    String uppercaseString = string.toUpperCase();
    collected.add(uppercaseString);
}

assertEquals(asList("A", "B", "HELLO"), collected);
```



```
List<String> collected = Stream.of("a", "b", "hello")
                               .map(string -> string.toUpperCase())  ❶
                               .collect(toList());

assertEquals(asList("A", "B", "HELLO"), collected);
```

Streams API Common Operations (Cont.)



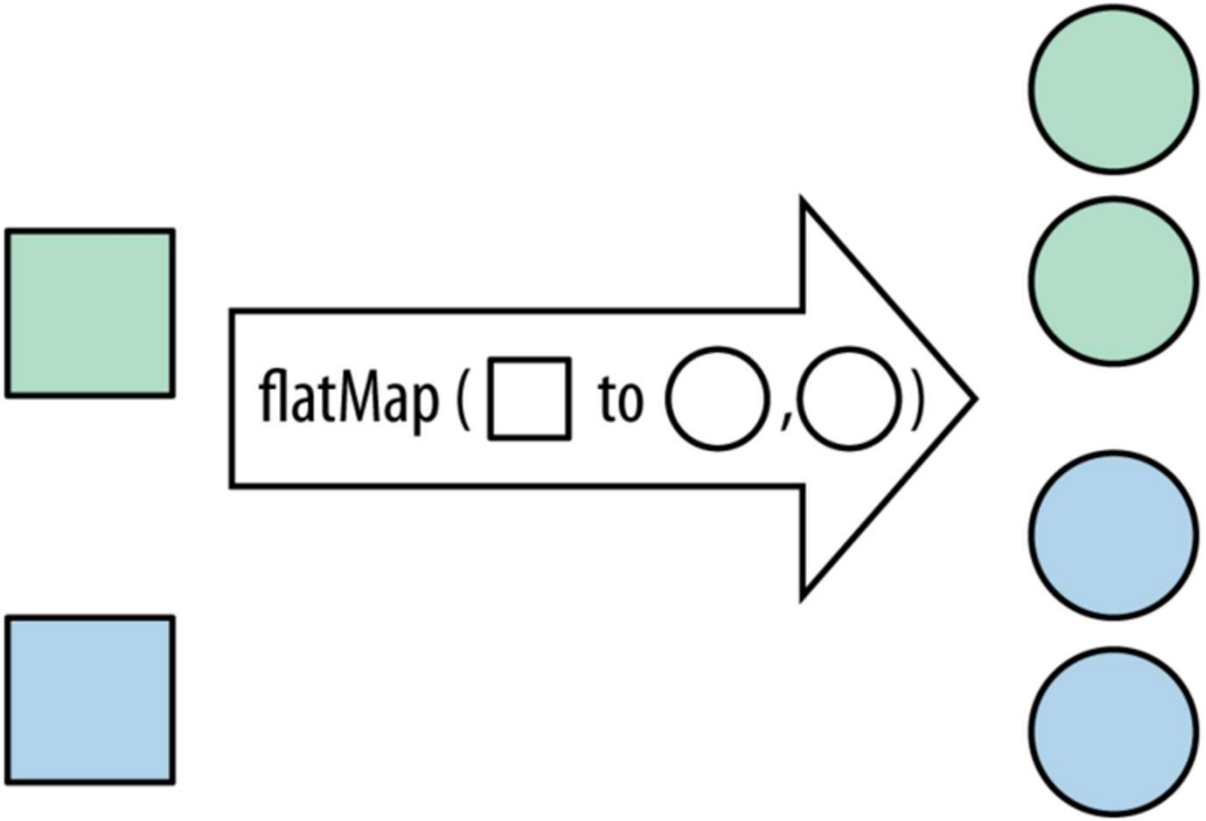
The `map` operation

Streams API Common Operations (Cont.)

- Intermediate Operations
  - **`flatMap`**: Replace a value with a Stream and concatenate all the streams together.

```
List<Integer> together = Stream.of(asList(1, 2), asList(3, 4))
                               .flatMap(numbers -> numbers.stream())
                               .collect(toList());

assertEquals(asList(1, 2, 3, 4), together);
```

Streams API Common Operations (Cont.)



The `flatMap` operation

Streams API Common Operations (Cont.)

- Intermediate Operations
  - **`sorted`**: Sorts elements based on a comparator.

```
1   List<String> sorted = list.stream()
2                             .sorted()
3                             .collect(Collectors.toList());
```

Streams API Common Operations (Cont.)

- Terminal Operations
  - **`forEach`**: Performs an action for each element.

```
1    list.stream().forEach(System.out::println);
```

Streams API Common Operations (Cont.)

- Terminal Operations
  - **`collect`**: Collects the elements into a collection or another container.

```
1   List<String> collected = list.stream()
2                             .collect(Collectors.toList());
```

```
List<String> collected = Stream.of("a", "b", "c")  ❶
                               .collect(Collectors.toList());  ❷

assertEquals(Arrays.asList("a", "b", "c"), collected);  ❸
```

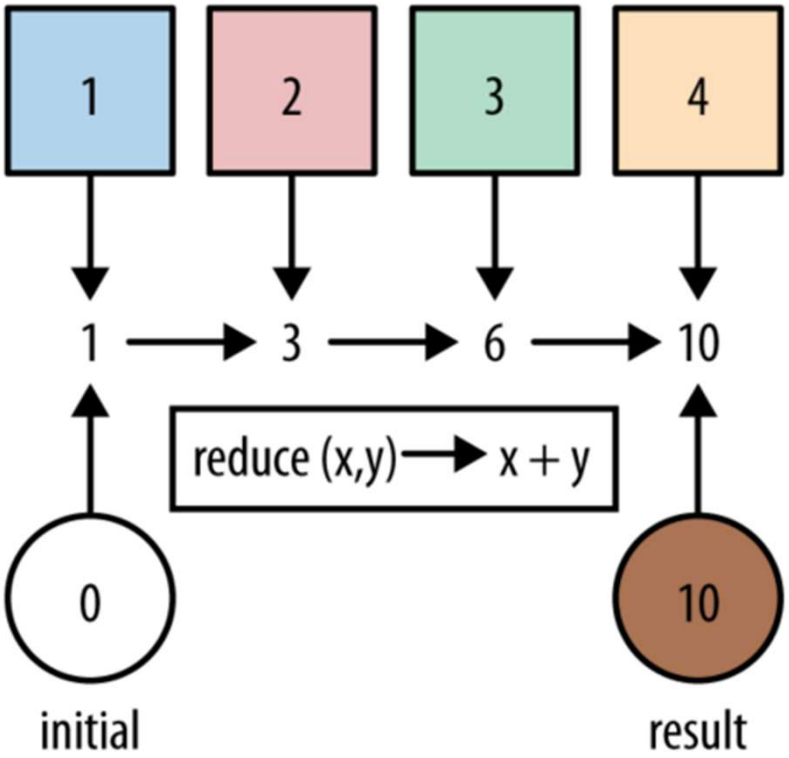
Streams API Common Operations (Cont.)

- Terminal Operations
  - **`reduce`**: Reduces the elements to a single value using an accumulator.

```
1    int sum = intStream.reduce(0, Integer::sum);
```

```
int count = Stream.of(1, 2, 3)
                  .reduce(0, (acc, element) -> acc + element);

assertEquals(6, count);
```

Streams API Common Operations (Cont.)



The `reduce` operation

Streams API Common Operations (Cont.)

- Parallel Streams
  - Streams can be processed in parallel to improve performance for large data sets.

```
1   List<String> collected = list.parallelStream()
2                                 .filter(s -> s.startsWith("a"))
3                                 .collect(Collectors.toList());
```

Streams API Example

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamExample {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("hello", "world", "functional", "programming", "java");

        // Convert words to uppercase, filter those starting with 'f', and collect into a list
        List<String> filteredWords = words.stream()
                                          .map(String::toUpperCase)
                                          .filter(word -> word.startsWith("F"))
                                          .collect(Collectors.toList());

        // Print the result
        filteredWords.forEach(System.out::println);
    }
}
```

Benefits of Streams API

- Conciseness
  - The Streams API provides a concise way to perform complex data manipulations with readable and maintainable code.
- Composability
  - Stream operations can be easily composed into pipelines, making it simple to build complex data processing tasks.

Benefits of Streams API (Cont.)

- Efficiency
  - Streams support lazy evaluation, which can optimize performance by avoiding unnecessary computations.
  - Parallel streams can further enhance performance by leveraging multiple CPU cores.
- Functional Programming
  - The Streams API encourages the use of functional programming principles, such as immutability and statelessness, leading to safer and more predictable code.