Generics, Lambda Expressions & Stream API

Advanced Java Programming

Masoud Hamad

State University of Zanzibar (SUZA)

2025/2026 Academic Year

Outline

Why Generics?

Without Generics (Java 1.4 and earlier):

- Collections stored `Object` references
- Required explicit casting on retrieval
- Type errors discovered only at **runtime**
- ClassCastException was common

With Generics (Java 5+):

- Type safety enforced at **compile time**
- No casting needed
- Code reuse with type parameters
- Bugs caught early during development

Key Benefit

Generics enable *stronger type checks at compile time*, eliminating the need for casts and reducing the risk of `ClassCastException` at runtime.

Before and After Generics

Before Generics — unsafe and verbose:

```
List list = new ArrayList();
list.add("Hello");
String s = (String) list.get(0);  // explicit cast required
list.add(42);                     // no compile error!
String t = (String) list.get(1);  // ClassCastException at runtime
```

After Generics — type-safe and clean:

```
List<String> list = new ArrayList<>();
list.add("Hello");
String s = list.get(0);     // no cast needed
// list.add(42);            // compile-time error!
```

Generic Classes

A generic class declares one or more **type parameters** in angle brackets:

```
public class Box<T> {
    private T content;

    public void set(T content) {
        this.content = content;
    }

    public T get() {
        return content;
    }
}
```

Usage:

```
Box<String> stringBox = new Box<>();
stringBox.set("Generics");
String value = stringBox.get();  // no cast

Box<Integer> intBox = new Box<>();
intBox.set(42);
int num = intBox.get();          // auto-unboxing
```

Generic Class with Multiple Type Parameters

```
public class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }

    @Override
    public String toString() {
        return "(" + key + ", " + value + ")";
    }
}
```

```
Pair<String, Integer> student = new Pair<>("Ali", 95);
System.out.println(student);            // (Ali, 95)
System.out.println(student.getKey());   // Ali
```

Generic Methods

A generic method declares its own type parameter(s), independent of the class:

```
public class Utility {
    // Generic method to print any array
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.print(element + " ");
        }
        System.out.println();
    }

    // Generic method that returns a value
    public static <T> T getFirst(List<T> list) {
        return list.isEmpty() ? null : list.get(0);
    }
}
```

```
Integer[] nums = {1, 2, 3, 4, 5};
String[] names = {"Ali", "Fatma", "Said"};
Utility.printArray(nums);     // 1 2 3 4 5
Utility.printArray(names);    // Ali Fatma Said
```

Bounded Type Parameters

Use `extends` to restrict which types can be used as arguments:

```
// T must implement Comparable<T>
public static <T extends Comparable<T>> T findMax(T[] array) {
    T max = array[0];
    for (T element : array) {
        if (element.compareTo(max) > 0) {
            max = element;
        }
    }
    return max;
}
```

```
Integer[] nums = {3, 7, 1, 9, 4};
System.out.println(findMax(nums));    // 9

String[] words = {"banana", "apple", "cherry"};
System.out.println(findMax(words));  // cherry
```

Multiple Bounds

`<T extends Comparable<T> & Serializable>` — T must satisfy all bounds (class first,

Wildcards

Wildcards (?) represent an *unknown type* and control what you can read/write:

Upper Bounded

`? extends T`

Read as T, cannot write.
"Producer" — produces T values.

```
void sum(List<? extends
         Number> list) {
    double s = 0;
    for (Number n : list)
        s += n.doubleValue()
            ;
}
```

Lower Bounded

`? super T`

Can write T, reads as Object.
"Consumer" — consumes T values.

```
void addNums(List<? super
             Integer> list)
                 {
    list.add(1);
    list.add(2);
    list.add(3);
}
```

Unbounded

?

Read as Object, cannot write.
Read-only access.

```
void print(List<?> list) {
    for (Object o : list)
        System.out.println(o
            );
}
```

PECS Principle

Producer **E**xtends, **C**onsumer **S**uper — use `extends` when you only read, `super` when you only write.

Type Erasure

- Java generics are implemented via **type erasure** — generic type information is removed at compile time
- The compiler replaces type parameters with their bounds (or `Object` if unbounded)
- Inserts casts where necessary to maintain type safety

What you write:

```
Box<String> box = new Box<>();
box.set("Hello");
String s = box.get();
```

After erasure:

```
Box box = new Box();
box.set("Hello");
String s = (String) box.get();
```

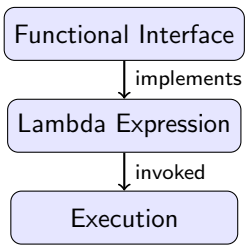
Consequences of Type Erasure

- Cannot use `new T()` or `new T[10]`
- Cannot use `instanceof` with generic types
- `List<String>` and `List<Integer>` are the same class at runtime

Functional Interfaces

A **functional interface** has exactly *one abstract method* (SAM — Single Abstract Method):

- Annotated with `@FunctionalInterface` (optional but recommended)
- Can have default and static methods
- Foundation for lambda expressions
- Lambda provides an implementation of the single abstract method



Lambda Expression Syntax

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

```java
// No parameters
Runnable r = () -> System.out.println("Hello!");

// One parameter (parentheses optional)
Consumer<String> greet = name -> System.out.println("Hi, " + name);

// Two parameters
Comparator<String> cmp = (a, b) -> a.length() - b.length();

// Multiple statements in body
BiFunction<Integer, Integer, Integer> max = (a, b) -> {
    if (a >= b) return a;
    else return b;
};
```

Rule of Thumb

If the body is a single expression, skip the braces and `return`. For multiple statements, use braces and explicit return.

Custom Functional Interface Example

```java
@FunctionalInterface
interface MathOperation {
    double operate(double a, double b);
}
```

```java
public class Calculator {
    public static void main(String[] args) {
        MathOperation add      = (a, b) -> a + b;
        MathOperation subtract = (a, b) -> a - b;
        MathOperation multiply = (a, b) -> a * b;
        MathOperation divide   = (a, b) -> a / b;

        System.out.println("10 + 5 = " + calculate(10, 5, add));
        System.out.println("10 - 5 = " + calculate(10, 5, subtract));
        System.out.println("10 * 5 = " + calculate(10, 5, multiply));
        System.out.println("10 / 5 = " + calculate(10, 5, divide));
    }

    static double calculate(double a, double b, MathOperation op) {
        return op.operate(a, b);
    }
}
```

Built-in Functional Interfaces (`java.util.function`)

Interface	Method	Input	Output
`Predicate<T>`	`test(T)`	T	boolean
`Function<T,R>`	`apply(T)`	T	R
`Consumer<T>`	`accept(T)`	T	void
`Supplier<T>`	`get()`	—	T
`UnaryOperator<T>`	`apply(T)`	T	T
`BiFunction<T,U,R>`	`apply(T,U)`	T, U	R
`BiPredicate<T,U>`	`test(T,U)`	T, U	boolean

```
Predicate<String> isLong = s -> s.length() > 5;
Function<String, Integer> toLen = String::length;
Consumer<String> printer = System.out::println;
Supplier<Double> random = Math::random;

System.out.println(isLong.test("Hello"));       // false
System.out.println(toLen.apply("Zanzibar"));    // 8
printer.accept("Lambda!");                       // Lambda!
System.out.println(random.get());               // 0.7234...
```

Predicate Composition

`Predicate<T>` supports chaining via `and()`, `or()`, and `negate()`:

```
Predicate<Integer> isPositive = n -> n > 0;
Predicate<Integer> isEven     = n -> n % 2 == 0;

Predicate<Integer> isPositiveAndEven = isPositive.and(isEven);
Predicate<Integer> isPositiveOrEven  = isPositive.or(isEven);
Predicate<Integer> isNonPositive     = isPositive.negate();

List<Integer> numbers = List.of(-4, -1, 0, 3, 6, 7, 10);

// Filter positive even numbers
numbers.stream()
       .filter(isPositiveAndEven)
       .forEach(System.out::println);  // 6, 10

// Filter non-positive numbers
numbers.stream()
       .filter(isNonPositive)
       .forEach(System.out::println);  // -4, -1, 0
```

Method References

Method references are shorthand for lambdas that call an existing method:

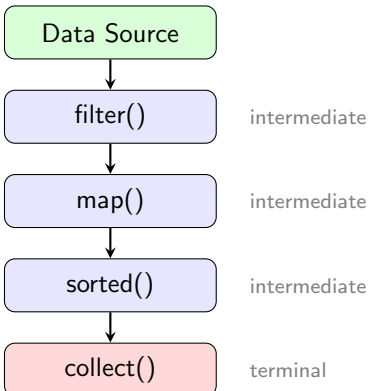
Type	Syntax	Lambda Equivalent
Static method	`ClassName::method`	`x -> Class.method(x)`
Instance method	`object::method`	`x -> obj.method(x)`
Arbitrary instance	`ClassName::method`	`x -> x.method()`
Constructor	`ClassName::new`	`x -> new Class(x)`

```
List<String> names = List.of("Ali", "Fatma", "Said", "Amina");

// Static method reference
names.forEach(System.out::println);

// Instance method of arbitrary object
names.stream().map(String::toUpperCase).forEach(System.out::println);

// Constructor reference
List<StringBuilder> builders = names.stream()
    .map(StringBuilder::new)
    .collect(Collectors.toList());
```

What is a Stream?

A **Stream** is a sequence of elements that supports *functional-style operations* on collections:

- **Not a data structure** — does not store elements
- **Pipeline** — source → intermediate ops → terminal op
- **Lazy evaluation** — intermediate operations are not executed until a terminal operation is invoked
- **Single use** — a stream can be consumed only once
- **Parallelizable** — easily switch to parallel execution



Creating Streams

```java
// From a Collection
List<String> names = List.of("Ali", "Fatma", "Said");
Stream<String> s1 = names.stream();

// From an array
String[] arr = {"Java", "Python", "C++"};
Stream<String> s2 = Arrays.stream(arr);

// Using Stream.of()
Stream<Integer> s3 = Stream.of(1, 2, 3, 4, 5);

// Using Stream.generate() (infinite stream)
Stream<Double> randoms = Stream.generate(Math::random).limit(5);

// Using Stream.iterate()
Stream<Integer> evens = Stream.iterate(0, n -> n + 2).limit(10);

// From a String
IntStream chars = "Hello".chars();

// From a file
Stream<String> lines = Files.lines(Path.of("data.txt"));
```

Intermediate Operations

Intermediate operations return a new stream and are **lazy**:

```
List<String> names = List.of("Ali", "Fatma", "Said", "Ali",
                             "Amina", "Fatma", "Hassan");

// filter: keep elements matching a predicate
names.stream().filter(n -> n.length() > 3)    // Fatma, Said, Amina, ...

// map: transform each element
names.stream().map(String::toUpperCase)       // ALI, FATMA, SAID, ...

// sorted: natural or custom order
names.stream().sorted()                        // Ali, Ali, Amina, ...
names.stream().sorted(Comparator.comparingInt(String::length))

// distinct: remove duplicates
names.stream().distinct()                      // Ali, Fatma, Said, Amina, Hassan

// limit and skip
names.stream().limit(3)                        // Ali, Fatma, Said
names.stream().skip(2)                         // Said, Ali, Amina, ...

// peek: debug without altering the stream
names.stream().peek(n -> System.out.println("Processing: " + n))
```

Terminal Operations

Terminal operations produce a result and **consume** the stream:

```
List<Integer> numbers = List.of(3, 7, 1, 9, 4, 7, 2);

// forEach: perform action on each element
numbers.stream().forEach(System.out::println);

// collect: gather elements into a collection
List<Integer> sorted = numbers.stream().sorted()
    .collect(Collectors.toList());

// reduce: combine elements into a single result
int sum = numbers.stream().reduce(0, Integer::sum);  // 33
Optional<Integer> max = numbers.stream().reduce(Integer::max);

// count
long count = numbers.stream().filter(n -> n > 3).count();  // 3

// anyMatch, allMatch, noneMatch
boolean hasNegative = numbers.stream().anyMatch(n -> n < 0);  // false

// findFirst, findAny
Optional<Integer> first = numbers.stream().filter(n -> n > 5).findFirst();
```

Stream Pipeline Example

```
record Student(String name, String department, double gpa) {}

List<Student> students = List.of(
    new Student("Ali",     "CS",     3.8),
    new Student("Fatma",   "CS",     3.5),
    new Student("Said",    "IT",     2.9),
    new Student("Amina",   "CS",     3.9),
    new Student("Hassan",  "IT",     3.2),
    new Student("Khadija","IT",      3.7)
);

// Find top 3 CS students by GPA
List<String> topCS = students.stream()
    .filter(s -> s.department().equals("CS"))    // only CS
    .sorted(Comparator.comparingDouble(Student::gpa).reversed())
    .limit(3)                                     // top 3
    .map(Student::name)                           // extract names
    .collect(Collectors.toList());

System.out.println(topCS);  // [Amina, Ali, Fatma]
```

Collectors — toList, toSet, toMap

```
List<String> names = List.of("Ali", "Fatma", "Said", "Amina");

// Collect to List
List<String> nameList = names.stream()
    .filter(n -> n.length() > 3)
    .collect(Collectors.toList());       // [Fatma, Said, Amina]

// Collect to Set (removes duplicates)
Set<String> nameSet = names.stream()
    .collect(Collectors.toSet());

// Collect to Map
Map<String, Integer> nameLengths = names.stream()
    .collect(Collectors.toMap(
        n -> n,              // key: name itself
        String::length       // value: length
    ));
// {Ali=3, Fatma=5, Said=4, Amina=5}
```

Collectors — groupingBy and partitioningBy

```
record Student(String name, String dept, double gpa) {}

List<Student> students = List.of(
    new Student("Ali", "CS", 3.8), new Student("Fatma", "IT", 3.5),
    new Student("Said", "CS", 2.9), new Student("Amina", "IT", 3.9)
);

// Group by department
Map<String, List<Student>> byDept = students.stream()
    .collect(Collectors.groupingBy(Student::dept));
// {CS=[Ali, Said], IT=[Fatma, Amina]}

// Group by department, count students
Map<String, Long> countByDept = students.stream()
    .collect(Collectors.groupingBy(Student::dept, Collectors.counting()));
// {CS=2, IT=2}

// Partition: split into two groups based on a predicate
Map<Boolean, List<Student>> partition = students.stream()
    .collect(Collectors.partitioningBy(s -> s.gpa() >= 3.5));
// {true=[Ali, Fatma, Amina], false=[Said]}
```

Collectors — joining, averagingDouble, summarizing

```
List<String> names = List.of("Ali", "Fatma", "Said", "Amina");

// joining: concatenate strings
String joined = names.stream()
    .collect(Collectors.joining(", ", "[", "]"));
// [Ali, Fatma, Said, Amina]

// averagingDouble
List<Student> students = /* ... */;
double avgGpa = students.stream()
    .collect(Collectors.averagingDouble(Student::gpa));

// summarizingDouble: min, max, avg, sum, count in one pass
DoubleSummaryStatistics stats = students.stream()
    .collect(Collectors.summarizingDouble(Student::gpa));
System.out.println("Average: " + stats.getAverage());
System.out.println("Max: " + stats.getMax());
System.out.println("Count: " + stats.getCount());
```

The reduce Operation In Depth

reduce combines stream elements into a single result:

```
List<Integer> numbers = List.of(1, 2, 3, 4, 5);

// With identity value
int sum = numbers.stream().reduce(0, (a, b) -> a + b);       // 15
int product = numbers.stream().reduce(1, (a, b) -> a * b);  // 120

// Without identity (returns Optional)
Optional<Integer> max = numbers.stream().reduce(Integer::max);
max.ifPresent(m -> System.out.println("Max: " + m));  // Max: 5

// String concatenation
List<String> words = List.of("Java", "is", "powerful");
String sentence = words.stream().reduce("", (a, b) -> a + " " + b).trim();
// "Java is powerful"

// Finding longest string
Optional<String> longest = words.stream()
    .reduce((a, b) -> a.length() >= b.length() ? a : b);
// Optional[powerful]
```

flatMap — Flattening Nested Structures

`flatMap` maps each element to a stream and then flattens the results:

```
List<List<String>> nested = List.of(
    List.of("Ali", "Fatma"),
    List.of("Said", "Amina"),
    List.of("Hassan")
);

// Flatten nested lists into a single stream
List<String> flat = nested.stream()
    .flatMap(Collection::stream)
    .collect(Collectors.toList());
// [Ali, Fatma, Said, Amina, Hassan]

// Split sentences into words
List<String> sentences = List.of("Hello World", "Java Streams");
List<String> words = sentences.stream()
    .flatMap(s -> Arrays.stream(s.split(" ")))
    .collect(Collectors.toList());
// [Hello, World, Java, Streams]
```

Parallel Streams

```
List<Integer> numbers = IntStream.rangeClosed(1, 1_000_000)
    .boxed().collect(Collectors.toList());

// Sequential
long start = System.nanoTime();
long seqSum = numbers.stream()
    .reduce(0, Integer::sum);
long seqTime = System.nanoTime() - start;

// Parallel
start = System.nanoTime();
long parSum = numbers.parallelStream()
    .reduce(0, Integer::sum);
long parTime = System.nanoTime() - start;

System.out.println("Sequential: " + seqTime / 1_000_000 + " ms");
System.out.println("Parallel:   " + parTime / 1_000_000 + " ms");
```

When to Use Parallel Streams

- Large datasets with CPU-intensive operations
- Operations must be **stateless** and **associative**

Comprehensive Example: Student Report

```
record Student(String name, String dept, double gpa, int age) {}

List<Student> students = List.of(
    new Student("Ali", "CS", 3.8, 22), new Student("Fatma", "CS", 3.5, 21),
    new Student("Said", "IT", 2.9, 23), new Student("Amina", "IT", 3.9, 20),
    new Student("Hassan", "CS", 3.2, 24), new Student("Khadija", "IT", 3.7, 22));

// Average GPA by department
Map<String, Double> avgGpa = students.stream()
    .collect(Collectors.groupingBy(Student::dept,
             Collectors.averagingDouble(Student::gpa)));

// Highest GPA student per department
Map<String, Optional<Student>> topPerDept = students.stream()
    .collect(Collectors.groupingBy(Student::dept,
             Collectors.maxBy(Comparator.comparingDouble(Student::gpa))));

// Names of students with GPA > 3.5, sorted alphabetically
String honors = students.stream()
    .filter(s -> s.gpa() > 3.5).sorted(Comparator.comparing(Student::name))
    .map(Student::name).collect(Collectors.joining(", "));
// "Ali, Amina, Khadija"
```

Summary

Generics

- Compile-time type safety
- Generic classes and methods
- Bounded type parameters
- Wildcards and PECS
- Type erasure at runtime

Lambda Expressions

- Functional interfaces
- Concise lambda syntax
- Predicate, Function, Consumer, Supplier
- Method references (`::`)
- Predicate composition

Stream API

- Declarative data processing
- Intermediate vs terminal ops
- Powerful Collectors
- reduce for aggregation
- Parallel streams

Next Steps

Practice by rewriting traditional loops using Streams and Lambdas. Combine Generics with Streams to build type-safe, reusable data processing pipelines.