Class-14 : Generics & Streams Introduction

by Pial Kanti Samadder

The Need for Generics

Let's imagine a scenario where we want to create a `list` in Java to store `Integer`.

We might try to write the following:

```
List list = new LinkedList();
list.add(new Integer(1));
Integer i = list.iterator().next();
```

Surprisingly, the compiler will complain about the last line. It doesn't know what data type is returned.

The compiler will require an explicit casting:

```
Integer i = (Integer) list.iterator.next();
```

A raw list can hold any object, so the compiler only guarantees `Object` as the return type. This forces explicit casts, which clutter code and may cause runtime errors.

Let's modify the first line of the previous code snippet:

```
List<Integer> list = new LinkedList<>();
```

By adding the diamond operator <> containing the type, we narrow the specialization of this list to only *Integer* type. In other words, we specify the type held inside the list. The compiler can enforce the type at compile time.

What are Generics

Generics are a way to tell the Java compiler **what type of objects are to be stored in a collection, or manipulated by a class or method, without actually specifying the exact type**.

- Generics let us **define classes, methods, and collections with a placeholder type**
- We use **angle brackets (`<>`)** to specify the type
- The type parameter can be reused throughout the class or method
- Allows code to be **type-safe, reusable, and flexible**

Example

```
List<String> names = new ArrayList<>();
```

Why Use Generics

Code that uses generics has many benefits over non-generic code:

- **Stronger type checks at compile time.**
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- **Elimination of casts.**
  The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

- **Enabling programmers to implement generic algorithms.**
  By using generics, programmers can implement generic algorithms that work on collections of different types, can be customized, and are type safe and easier to read.

Generic Types

A *generic type* is a generic class or interface that is parameterized over types.

A Simple Box Class

Begin by examining a non-generic `Box` class that operates on objects of any type. It needs only to provide two methods: set, which adds an object to the box, and get, which retrieves it:

```
public class Box {
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Since `Box` methods use `Object`, you can store anything (except primitives). But the compiler cannot check type usage. One part of the code may store an `Integer` while another stores a `String`, leading to runtime errors.

A Generic Version of the Box Class

A *generic class* is defined with the following format:

```
class name<T1, T2, ..., Tn> { /* ... */ }
```

The type parameter section, delimited by angle brackets (<>), follows the class name. It specifies the *type parameters* (also called *type variables*) T1, T2, ..., and Tn.

With this change, the Box class becomes:

```
public class Box<T> {
    // T stands for "Type"
    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

As you can see, all occurrences of Object are replaced by T. A type variable can be any **non-primitive** type you specify: any class type, any interface type, any array type, or even another type variable.

When you instantiate a `Box` object, you specify the type of `T`:

```
Box<Integer> integerBox = new Box<>();
integerBox.set(12345);
Integer someNumber = integerBox.get();
```

Raw Types

A *raw type* is **the name of a generic class or interface without any type arguments**. For example, given the generic Box class:

```
public class Box<T> {
    public void set(T t) { /* ... */ }
    // ...
}
```

To create a parameterized type of Box<T>, you supply an actual type argument for the formal type parameter T:

```
Box<Integer> intBox = new Box<>();
```

If the actual type argument is omitted, you create a raw type of Box<T>:

```
Box rawBox = new Box();
```

Raw types show up in legacy code because lots of API classes (such as the Collections classes) were not generic prior to JDK 5.0. When using raw types, you essentially get pre-generics behavior — a `Box` gives you Objects.

For backward compatibility, assigning a parameterized type to its raw type is allowed:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;             // OK
```

But if you assign a raw type to a parameterized type, you get a warning:

```
Box rawBox = new Box();          // rawBox is a raw type of Box<T>
Box<Integer> intBox = rawBox;    // warning: unchecked conversion
```

You also get a warning **if you use a raw type to invoke generic methods defined in the corresponding generic type**:

```
Box<String> stringBox = new Box<>();
Box rawBox = stringBox;
rawBox.set(8);  // warning: unchecked invocation to set(T)
```

Generic Methods

A generic method declaration can be called with arguments of different types. The compiler handles each method call appropriately based on the types of the arguments.

Syntax

```
<T> void myMethod( T argument ) {
  // Method body
}
```

- `<T>` before return type → declares type parameter
- Type parameter(s) can be used in **return type**, **arguments**, or **method body**

Example

```
public class GenericMethodExample {
  public static <T> void arrayPrint(T[] a) {
    for (T item : a) {
      System.out.println(item);
    }
  }

  public static void main(String[] args) {
    Integer[] nums = {0,1,2,3};
    String[] words = {"Hello","World"};
    Character[] chars = {'G','E','N','E','R','I','C'};

    arrayPrint(nums);
    arrayPrint(words);
    arrayPrint(chars);
  }
}
```

- Works with multiple types → `Integer`, `String`, `Character`
- One method, **many uses**

Bounded Type Parameters

There may be times when you want to **restrict the types** that can be used as type arguments in a parameterized type.

For example, a method that operates on numbers might only want to accept instances of `Number` or its subclasses. This is what *bounded type parameters* are for.

```
public class Box<T extends Number> {
    private T t;

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

    public void inspect(T u) {
        System.out.println("T: " + t.getClass().getName());
        System.out.println("U: " + u.getClass().getName());
    }

    public static void main(String[] args) {
        Box<Integer> box = new Box<>();
        box.set(123);

        box.inspect(45);    // Integer is allowed
        // box.inspect("Hi"); // String not allowed, compile-time error
    }
}
```

Multiple Bounds

- A type parameter can **extend a class and implement interfaces**
- Class must come **first**

```
class Person { String name; }
interface Payable { void paySalary(); }
interface Reportable { void generateReport(); }

class EmployeeProcessor<T extends Person & Payable & Reportable> {
    void process(T e) {
        System.out.println(e.name);
        e.paySalary();
        e.generateReport();
    }
}

class Manager extends Person implements Payable, Reportable {
    public void paySalary() { System.out.println("Salary paid."); }
    public void generateReport() { System.out.println("Report generated."); }
}

// Usage
EmployeeProcessor<Manager> processor = new EmployeeProcessor<>();
processor.process(new Manager());
```

- Only types that are **Person + Payable + Reportable** are allowed
- Ensures **compile-time type safety**

Java Stream

A Java *Stream* is a component that is capable of *internal iteration* of its elements, meaning it can iterate its elements itself.

Features

- A Stream is not a data structure; it just takes input from Collections, Arrays or I/O channels.
- Streams do not modify the original data; they only produce results using their methods.
- Intermediate operations (like filter, map, etc.) are lazy and return another Stream, so you can chain them together.
- A terminal operation (like collect, forEach, count) ends the stream and gives the final result.

Obtain a Stream

There are many ways to obtain a Java *Stream*. One of the most common ways to obtain a `Stream` is from a *Java Collection*. Here is an example of obtaining a `Stream` from a *Java List*:

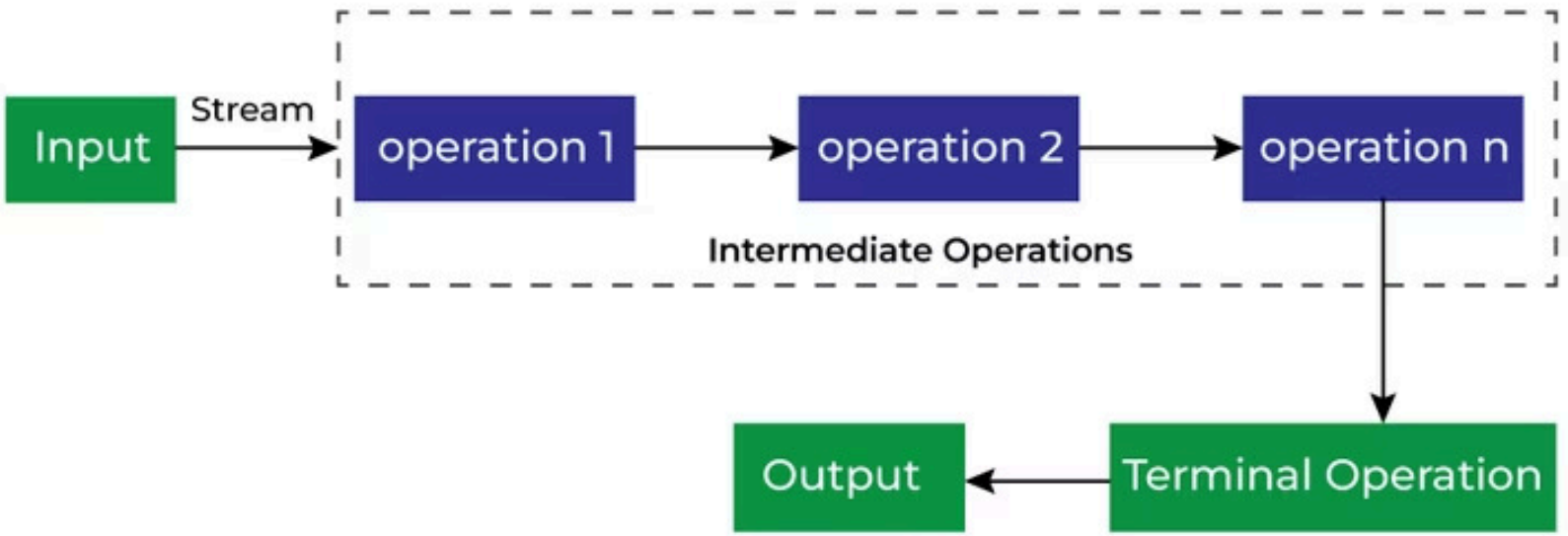
```
List<String> items = new ArrayList<String>();

items.add("one");
items.add("two");
items.add("three");

Stream<String> stream = items.stream();
```

Once created, the instance **will not modify its source,** therefore allowing the creation of multiple instances from a single source.

Different Operations On Streams

There are two types of Operations in Streams:

1. **Intermediate Operations**
2. **Terminal Operations**



Intermediate Operations

Intermediate Operations are the types of operations in which multiple methods are chained in a row.

Characteristics of Intermediate Operations

- Methods are chained together.
- Intermediate operations transform a stream into another stream.
- It enables the concept of filtering where one method filters data and passes it to another method after processing.

Important Intermediate Operations

filter()

The Java `Stream` `filter()` can be used to filter out elements from a Java `Stream`. The `filter` method takes a `Predicate` which is called for each element in the stream. If the element is to be included in the resulting `Stream`, the `Predicate` should return `true`. If the element should not be included, the `Predicate` should return `false`.

Here is an example of calling the Java `Stream` `filter()` method:

```
Stream<String> longStringsStream = stream.filter((value) -> {
    return value.length() >= 3;
});
```

map()

The Java `Stream` `map()` method converts (maps) an element to another object. For instance, if you had a list of strings it could convert each string to lowercase, uppercase, or to a substring of the original string, or something completely else. Here is a Java `Stream` `map()` example:

```
List<String> list = new ArrayList<String>();
Stream<String> stream = list.stream();

Stream<String> streamMapped = stream.map((value) -> value.toUpperCase());
```

Important Intermediate Operations

flatMap()

The `flatMap()` method maps a single element into multiple elements, effectively "flattening" nested structures. For example, it can convert a stream of strings into a stream of words. Each element is transformed into a stream, and `flatMap()` combines these streams into one.

```
List<String> titles = List.of(
    "One flew over the cuckoo's nest",
    "To kill a mockingbird",
    "Gone with the wind"
);

titles.stream()
      .flatMap(s -> Arrays.stream(s.split(" ")))
      .forEach(System.out::println);
```

Here, each string is split into words and flattened into a single stream. The terminal operation `forEach()` triggers the processing; without it, the flat mapping would not execute.

distinct()

The `distinct()` method returns a new stream containing only the **unique elements** from the original stream, removing any duplicates.

```
List<String> strings = List.of("one", "two", "three", "one");

List<String> distinctStrings = strings.stream()
                                      .distinct()
                                      .collect(Collectors.toList());

System.out.println(distinctStrings); // [one, two, three]
```

In this example, the duplicate `"one"` is removed, so the resulting list contains only `"one"`, `"two"`, and `"three"`.

Terminal Operations

Terminal operations are operations on a stream that **produce a result or a side-effect** and **end the stream pipeline**.

Characteristics of Terminal Operations

- **End the stream pipeline**; no further operations can be chained
- **Trigger the internal iteration** of the stream
- Produce a **final result** or a **side-effect**

Important Terminal Operations

collect()

The `collect()` method is a **terminal operation** that starts the stream's internal iteration and gathers elements into a **collection or object**. It takes a `Collector` as a parameter.

```
List<String> titles = List.of(
    "One flew over the cuckoo's nest",
    "To kill a mockingbird",
    "Gone with the wind"
);

List<String> upperCaseTitles = titles.stream()
                               .map(String::toUpperCase)
                               .collect(Collectors.toList());

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

- In this example, `Collectors.toList()` collects the stream elements into a **List**.
- The `Collectors` class provides **ready-made Collector implementations** for common operations.

forEach()

The `forEach()` method is a **terminal operation** that iterates over each element in the Stream and applies a `Consumer`. It returns `void`.

```
List<String> list = List.of("one", "two", "three", "one");

list.stream().forEach(e -> System.out.println(e));
```

Important Terminal Operations

anyMatch()

The `anyMatch()` method is a **terminal operation** that tests whether **any element** of the stream matches a given `Predicate`. Returns `true` if a match is found, otherwise `false`.

```
List<String> list = List.of(
    "One flew over the cuckoo's nest",
    "To kill a muckingbird",
    "Gone with the wind"
);

boolean result = list.stream().anyMatch(s -> s.startsWith("One"));

System.out.println(result); // true
```

allMatch()

The `allMatch()` method is a **terminal operation** that checks if **all elements** in the stream match a given `Predicate`. Returns `true` only if every element matches.

```
List<String> list = List.of(
    "One flew over the cuckoo's nest",
    "To kill a muckingbird",
    "Gone with the wind"
);

boolean result = list.stream().allMatch(s -> s.startsWith("One"));

System.out.println(result); // false
```

Important Terminal Operations

noneMatch()

The `noneMatch()` method is a **terminal operation** that checks if **no elements** in the stream match the given `Predicate`. Returns `true` if none match, otherwise `false`.

```
List<String> list = List.of("abc", "def");

boolean result = list.stream().noneMatch(s -> s.equals("xyz"));

System.out.println(result); // true
```

count()

The `count()` method is a **terminal operation** that returns the number of elements in a stream.

```
List<String> words = List.of("apple", "banana", "avocado", "cherry");

long count = words.stream()
        .filter(w -> w.startsWith("a"))
        .count();

System.out.println(count); // 2
```

Important Terminal Operations

findAny()

The Java `Stream` `findAny()` method can find a single element from the Stream. The element found can be from anywhere in the `Stream`.

```
List<String> stringList = List.of("one", "two", "three", "one");

Optional<String> anyElement = stringList.stream()
        .filter(s -> s.length() > 3)
        .findAny();

anyElement.ifPresent(System.out::println);
```

findFirst()

The `findFirst()` method is a **terminal operation** that returns an `Optional` containing the **first element** in the stream, if present.

```
List<String> list = List.of("one", "two", "three", "one");

Optional<String> first = list.stream().findFirst();

first.ifPresent(System.out::println); // one
```

Important Terminal Operations

reduce()

The Java `Stream` `reduce()` method is a terminal operation that can reduce all elements in the stream to a single element.

```
List<String> stringList = List.of(
            "One flew over the cuckoo's nest",
            "To kill a muckingbird",
            "Gone with the wind");

Optional<String> reduced = stringList.stream()
        .reduce((value, combinedValue) -> combinedValue + ", " + value);

reduced.ifPresent(System.out::println);
```

Lambda Expressions

Lambda Expressions were added in Java 8.

A **lambda expression** is a short block of code that takes in parameters and returns a value. Lambdas look similar to methods, but they do not need a name, and they can be written right inside a method body.

Syntax

The simplest lambda expression contains a single parameter and an expression:

```
parameter -> expression
```

To use more than one parameter, wrap them in parentheses:

```
(parameter1, parameter2) -> expression
```

Simple expressions must return a value immediately. They cannot contain multiple statements, such as loops or `if` conditions. To do more complex work, use a code block with curly braces. If the lambda should return a value, use the `return` keyword:

```
(parameter1, parameter2) -> {
  // code block
  return result;
}
```

Predicate

Represents a boolean-valued function of one argument.

```
Predicate<Integer> isEven = number -> number % 2 == 0;
System.out.println(isEven.test(4)); // Output: true
```