EnumJ Design

Contents

[2 Introduction 1](#_Toc427005548)

[3 Limitations of Java 8 Streams 1](#_Toc427005549)

[4 EnumJ Design Goals 2](#_Toc427005550)

# Introduction

This document contains a description of the design decisions for **EnumJ**, a **Java 8** library for highly composable iterators. The presentation herein assumes the reader is familiar with the **Iterator** and **Iterable** interfaces as well as with **Java 8** constructs like streams, lambda expressions, functional interfaces and default methods.

# Limitations of Java 8 Streams

Streams are a great addition to Java as they provide the ability to apply transforming operations upon entire collections thus enabling the developer to work at a higher level of abstraction. In addition to that, **Java 8** streams offer seamless parallelism empowering the programmer with idioms for easy, expressive and safe concurrent computing.

Despite their power, streams suffer from limitations that hinder the full exploitation of their otherwise beautiful, declarative nature. Among limitations are:

* Limited scalability in terms of compositions
* Lack of repeatability
* Lack of shareability
* Lack of caching
* Mandatory eagerness (with few exceptions)
* Lack of late binding
* Lack of fault tolerance

The following table shows the consequences of these limitations.

|  |  |  |
| --- | --- | --- |
| Limitation | Consequence | Details |
| Limited scalability | No basis for data generators defined at runtime | Unsuiotable for constraint processing, logic programming, random path simulations, etc |
| Lack of repeatability | Poor abstraction for composed computation | Such an ability would allow to construct “code” at runtime and pass it on for execution, much like higher-order functions |
| Lack of shareability | Prevents streams from participation to computational topologies | Such an ability would allow to compose “complex”, “non-linear” “code” at runtime and pass it on for execution |
| Lack of caching | Forces the programmer to fetch eagerly in order to prevent duplicated execution | Caching while maintaining fluency of composition is essential for efficiency |
| Mandatory eagerness | Unable to define potentially infinite graphs of inter-dependent generators | Lazy composition is essential for self-referring, potentially infinite data generators |
| Lack of late binding | Non-serializable by nature, hence non-distributable | Distributed execution requires late-binding of data source |
| Lack of fault tolerance | Unsuitable for large, expensive computations where continuing past minor errors is a must | Long, expensive algorithms must continue beyond small errors, they cannot afford to stop in the middle |

**EnumJ** tries to address these limitations.

# EnumJ Design Goals

The major goal of **EnumJ** is to address the limitations exhibited by streams. The following table shows how each limitation is being addressed.

|  |  |  |
| --- | --- | --- |
| Stream limitation | How is addressed by EnumJ | Details |
| Limited scalability | Massive scalability on most operations | Transforms calls into iterations by maintaining an internal pipeline of element sources and element processors |
| Lack of repeatability | Enumerable<E> | Just as **Enumerator<E>** is a composable **Iterator<E>**, **Enumerable<E>** is a composable **Iterable<E>** that can spawn **Enumerable<E>** instances repeatedly |
| Lack of shareability | ShareableEnumerator<E>, Enumerable<E> | Two ways of sharing compositions |
| Lack of caching | CachedEnumerable<E> | Subsequent spawned enumerators reuse cached elements when enumerating |
| Mandatory eagerness | Enumera(tor/ble).ofLazyXYZ(…) | Enumerators based on lazily specified sequences |
| Lack of late binding | Enumera(tor/ble)  .ofLateBinding(Class<T>) | Enumerators that can compose without a source. The source can be bound later, right before enumeration |
| Lack of fault tolerance | Enumera(tor/ble)  .asTolerant(Consumer<Exception>) | Enumerators that can recover from errors by passing the exception to a handler and continuing |