

FLORIN RADU

CONCURRENT PROGRAMMING IN JAVA

From Traditional Threads to Project Loom



THREADS • SYNCHRONIZATION • STRUCTURED CONCURRENCY

CONCURRENT PROGRAMMING IN JAVA

From Traditional Threads to Project Loom

FLORIN RADU

2026

Copyright © 2026 Florin Radu

All rights reserved.

First English edition, 2026.

No part of this book may be reproduced or distributed in any form without prior written permission from the author, except for brief quotations in reviews, uses permitted by law, and code examples explicitly released under a separate software license.

Java and related marks are trademarks of Oracle and/or its affiliates. This book is an independent publication and is not affiliated with or endorsed by Oracle.

Every effort has been made to ensure technical accuracy. Software platforms evolve, however, and readers should consult the official Java documentation for API details that may change after publication.

Companion Source Code

The complete, runnable Java examples accompanying this book are available at:

https://github.com/florin2radu/concurrent_programming_in_java

The repository is organized by chapter and section and contains runnable versions of the examples presented throughout the book.

In addition to the examples presented in the book, the repository includes a separate package containing more advanced, real-world examples that combine multiple concurrency mechanisms in practical scenarios.

Unless otherwise stated, the source code in the companion repository is licensed under the Apache License 2.0.

Use of Generative AI in Translation and Editorial Review

This English edition was translated from the original Romanian edition using generative artificial intelligence tools and was subsequently reviewed, revised, and extensively edited by the author.

Generative AI tools were also used in selected sections to support drafting and reformulation, linguistic and technical editing, terminology consistency, identification of potential technical inconsistencies, and review and correction of selected Java code examples. They were also used to support editorial consistency in headings, code listings, tables, and captions.

Artificial intelligence tools were not used as independent bibliographic sources and did not replace the official Java documentation or the technical and scientific references cited in this book. Technical claims, code examples, terminology, and editorial changes were reviewed by the author.

The structure of the book, the selection and organization of topics, the examples, explanations, pedagogical approach, and final editorial decisions remain the responsibility of the author. The author assumes full responsibility for the content of this edition.

Preface

Concurrent programming has always required two kinds of understanding: familiarity with the APIs and a sound mental model of what actually happens when multiple tasks make progress at the same time. Java has evolved from direct thread management to executors, concurrent collections, asynchronous composition, virtual threads, and structured concurrency, yet the underlying questions remain the same: What state is shared? What ordering is guaranteed? What can run in parallel? What can go wrong?

This book follows that evolution deliberately. It begins with the foundations of concurrency and parallelism, introduces the functional features used throughout modern Java, then develops the traditional threading model, synchronization, `java.util.concurrent`, common concurrency failures, reusable design patterns, and finally the ideas introduced by Project Loom.

The emphasis is practical. Examples are kept small enough to understand, but they are chosen to expose behaviors that matter in real systems. The reader is expected to have a working knowledge of Java; no advanced background in concurrency theory is required.

Contents

1. Introduction.....	1
1.1. Why Concurrent Programming?.....	1
1.2. What Does “Concurrency” Mean in Java?.....	1
1.3. The Evolution of Concurrency Support in Java.....	2
1.4. Goals of This Book.....	3
1.5. Who Is This Book For?.....	4
2. Concurrency Fundamentals.....	5
2.1. What Is Concurrency?.....	5
2.2. Concurrency vs. Parallelism.....	5
2.3. Process vs. Thread.....	6
2.4. Benefits of Concurrent Programming.....	8
2.5. Challenges and Risks of Concurrent Programming.....	10
2.6. Concurrency Models.....	13
2.7. Conclusion.....	18
3. Lambda Expressions and the Stream API.....	19
3.1. From Anonymous Classes to Lambda Expressions.....	19
3.2. Functional Interfaces.....	20
3.3. Lambda Expressions and Captured Variables.....	21
3.4. Method References and Equivalent Lambda Expressions.....	22
3.5. Introduction to the Stream API.....	24
3.6. Intermediate and Terminal Stream Operations.....	26
3.7. Practical Stream API Examples.....	28
3.8. Optional and Missing Values.....	29
3.9. Parallel Streams.....	32
3.10. Conclusion.....	33
4. Threads in Java - The Traditional Model.....	34
4.1. Introduction.....	34
4.2. How Threads Execute.....	35
4.3. Creating and Starting Threads.....	37
4.4. Common Thread Methods.....	42
4.5. Mutual Exclusion with synchronized.....	47
4.6. Thread Cooperation with wait(), notify(), and notifyAll().....	56
4.7. volatile Variables - Visibility Between Threads.....	60
4.8. Thread Scheduling.....	64
5. Modern Concurrency Mechanisms in Java (java.util.concurrent).....	74
5.1. Introduction to java.util.concurrent.....	74
5.2. Locks and Conditions.....	76
5.3. Synchronization with Semaphores and Barriers.....	83

5.4. Atomic Operations and Lock-Free Programming.....	90
5.5. Concurrent Data Structures - Thread-Safe Collections.....	96
5.6. Synchronizing Data Exchanges.....	102
5.7. Executor Framework - Modern Thread Management.....	107
5.8. CompletableFuture - Modern Asynchronous Programming in Java.....	116
5.9. Best Practices for Using java.util.concurrent.....	127
5.10. Conclusions.....	130
6. Classic Problems in Concurrent Programming.....	133
6.1. Race Conditions.....	133
6.2. Deadlocks.....	139
6.3. Livelock.....	146
6.4. Thread Starvation.....	150
6.5. Other Problems in Concurrent Programming.....	153
7. Design Patterns for Concurrent Programming.....	157
7.1. Coordination Patterns.....	158
7.2. Execution Patterns.....	165
7.3. Messaging Patterns.....	170
7.4. Structural Patterns.....	175
7.5. Conclusions.....	179
8. Virtual Threads and Structured Concurrency (Project Loom).....	181
8.1. Introduction: The Shift Brought by Project Loom.....	181
8.2. What Are Virtual Threads?.....	186
8.3. Benefits and Limitations of Virtual Threads.....	189
8.4. Creating Virtual Threads.....	193
8.5. Structured Concurrency.....	197
9. General Conclusions: Choosing the Right Tool for Modern Java Concurrency.....	215
9.1. From Traditional Threads to Modern Concurrency.....	215
9.2. There Is No Universal Mechanism.....	216
9.3. When to Use Traditional Mechanisms.....	216
9.4. When to Use Modern java.util.concurrent Tools.....	218
9.5. When to Choose ExecutorService.....	220
9.6. When to Choose CompletableFuture.....	221
9.7. When to Choose Virtual Threads.....	221
9.8. When to Choose Structured Concurrency.....	222
9.9. A Practical Decision Rule.....	223
9.10. Common Mistakes in Concurrent Applications.....	224
9.11. Where Is Java Heading?.....	225
9.12. Final Conclusion.....	226
References.....	227

1. Introduction

1.1. Why Concurrent Programming?

For many years, one of the main ways to make a program faster was to increase the processor clock frequency. Today, the situation is very different. Modern processors contain an increasing number of cores - four, eight, or even dozens - and can execute multiple operations in parallel [1, 2, 9].

Simply having more cores, however, does not automatically make an application faster. Developers must structure software so that it can actually exploit the available hardware parallelism. This is where concurrent programming becomes essential.

Consider an application that processes large files or calls several external services. Independent operations should not necessarily be forced to wait for one another. By using multiple threads or higher-level concurrency mechanisms, an application can make progress on several tasks at once, improving throughput and responsiveness.

1.2. What Does “Concurrency” Mean in Java?

Concurrency is broader than parallelism [1, 2].

Parallelism means that multiple operations are physically executing at the same instant, whereas concurrency is the organization of multiple flows of work whose execution overlaps in time. Java supports both approaches.

Through `Thread`, `ExecutorService`, and specialized data structures such as `ConcurrentHashMap` and `BlockingQueue`, Java provides mechanisms for coordinating many tasks without forcing them into a single sequential flow. In modern software - web servers, games, data-processing applications, and REST services - concurrency is often unavoidable.

1.3. The Evolution of Concurrency Support in Java

Early Java releases, around 1996, provided the `Thread` class and the `Runnable` interface as the main primitives for concurrent programming. They were useful, but low-level. As applications grew in complexity, manual thread management quickly became a source of errors [1, 3].

Java 5, released in September 2004, was a major milestone in the language. Alongside features such as generics, the enhanced for loop, autoboxing and unboxing, annotations, and variable-arity methods, it introduced the `java.util.concurrent` package, which significantly changed the way concurrent Java code could be written [16]. Among its most important additions were:

- `ExecutorService` - a higher-level mechanism for managing task execution with thread pools and work queues;
- explicit locks such as `ReentrantLock` and `ReadWriteLock`, which provide finer control than `synchronized`;
- `Semaphore` - limits the number of threads that may access a shared resource at the same time, which is useful for connection limits and database access;
- `CountDownLatch` - allows one or more threads to wait until a set of operations performed by other threads has completed;
- `CyclicBarrier` - coordinates a group of threads so that each waits at a common barrier until all participants arrive, after which they can proceed;
- `Exchanger` - allows two threads to exchange data in a safe, `synchronized` manner;
- thread-safe data structures such as `ConcurrentHashMap`, `CopyOnWriteArrayList`, `CopyOnWriteArraySet`, and `LinkedBlockingQueue`;
- `Future` and `Callable` - abstractions for asynchronous computations that produce results.

Java 7 introduced the Fork/Join Framework for decomposing and parallelizing larger tasks. Java 8, released in March 2014, added a higher level of abstraction through parallel streams and `CompletableFuture`, integrating concurrency more deeply with the functional style of modern Java [6, 7, 34, 35, 37].

Another major shift came with virtual threads, introduced in Java 21 in September 2023. As part of Project Loom, virtual threads greatly reduce the cost of managing large numbers of concurrent threads and make it practical to use a straightforward thread-per-task style for applications dominated by blocking operations [44, 45].

Taken together, these developments show several major shifts in Java concurrency: from direct thread management, to executors and concurrent data structures, then to functional and asynchronous abstractions, and finally to virtual threads and the broader ideas of Project Loom.

1.4. Goals of This Book

This book is not merely a collection of definitions. It is a practical guide for anyone who wants to write code that:

- uses multicore hardware effectively;
- remains correct and stable, avoiding deadlocks, synchronization errors, and lost data;
- is easy for a development team to understand and maintain.

Through concrete examples, you will learn how to:

- use synchronization mechanisms correctly, including `synchronized`, `volatile`, and explicit locks;
- understand how the Java Memory Model works;
- implement safe patterns for concurrent and parallel execution;
- test and debug concurrent code;
- design scalable architectures using `ExecutorService`, `Future`, and parallel streams.

1.5. Who Is This Book For?

This book is for you if:

- you have already written Java applications and want to make them faster or more efficient;
- you work with enterprise applications, microservices, or systems that handle many simultaneous requests;
- you are a student who wants to understand in depth how threads and modern Java concurrency work;
- you have encountered terms such as deadlock, race condition, or thread pool and want to understand how to handle them correctly.

The book does not assume an advanced theoretical background. A solid foundation in Java and a willingness to learn through practice are enough.

2. Concurrency Fundamentals

2.1. What Is Concurrency?

Concurrency is the ability of a program to manage multiple tasks whose execution overlaps in time. It does not necessarily mean that the tasks run at exactly the same instant; rather, it means that the program can make progress on several activities without waiting for one to finish before starting the next.

A simple example is a desktop application that saves a file in the background while the user continues working. Another is a web server that can respond to hundreds of requests concurrently, with each request handled by its own execution context [1, 2, 8, 9].

Concurrency is therefore a way of organizing work so that a system can make progress on several activities at once. When the hardware and operating system allow multiple threads to run on different processor cores, some of those activities may also execute in true parallel.

On a multicore processor, for example, the JVM and operating system may schedule different Java threads on different cores, allowing two or more tasks to execute simultaneously rather than merely being interleaved.

On a single-core processor, by contrast, concurrency is achieved through rapid context switching: the operating system temporarily suspends one runnable thread and lets another execute, creating the appearance that several tasks are progressing at the same time.

2.2. Concurrency vs. Parallelism

The two terms are often used interchangeably, but they describe different ideas.

Concurrency means that multiple tasks overlap in time, although they do not necessarily execute at the same instant [1, 8, 9].

Parallelism means that multiple operations are physically executing at the same time, typically on different processor cores or processors.

A useful analogy is a waiter serving several tables. One waiter switching between tables illustrates concurrency; several waiters serving different tables at the same time illustrates parallelism.

Because the developer does not directly control how the operating system maps threads to processor cores, and because an application may have far more runnable threads than available cores, Java programming is primarily concerned with concurrency: coordinating independent activities correctly and safely, whether or not they happen to execute in parallel at a particular moment.

The JVM and the operating system determine when and where platform threads execute through thread-scheduling mechanisms. From the developer's perspective, the key concern is therefore correct coordination rather than direct control over physical parallel execution.

2.3. Process vs. Thread

A process is an instance of a program in execution. Each process has its own address space and operating-system resources and is isolated from other processes (see Figure 1) [3, 10, 11].

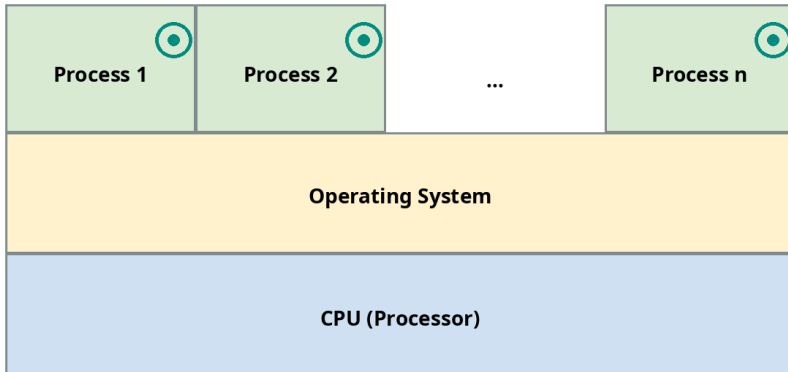


Figure 1. Processes managed by the operating system

A thread, by contrast, is a smaller unit of execution that exists within a process (see Figure 2). Threads belonging to the same process share the process memory and resources, while each thread has its own execution stack and can execute a different sequence of instructions [13].

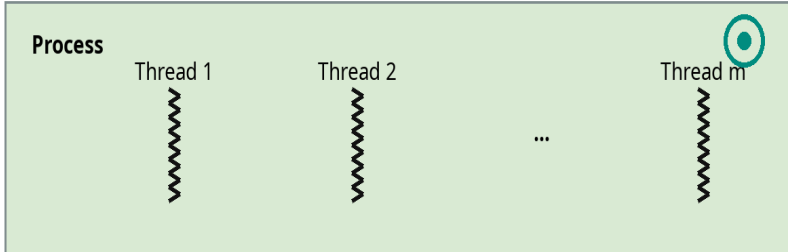


Figure 2. Threads within a process

Threads make communication and data sharing inexpensive compared with separate processes and can usually be created and scheduled with less overhead. The trade-off is that shared memory introduces subtle failure modes: unsynchronized access can lead to race conditions, deadlocks, visibility problems, and other errors that are difficult to reproduce.

The following minimal example creates two threads:

```
public class HelloThread extends Thread {
    @Override
    public void run() {
        System.out.println("Hello from thread: " +
            Thread.currentThread().getName());
    }

    public static void main(String[] args) {
        new HelloThread().start(); // typically Thread-0
        new HelloThread().start(); // typically Thread-1
    }
}
```

The `HelloThread` class extends the predefined `Thread` class and overrides `run()` with the logic that each thread will execute. `Thread.currentThread().getName()` returns the name of the thread that is currently running. Although the first `HelloThread` instance is created and started before the second, the order in which their messages appear is not guaranteed and may vary from one run to another. This nondeterminism is a fundamental characteristic of concurrent execution.

2.4. Benefits of Concurrent Programming

Why introduce multiple threads and the additional complexity that comes with them? Because concurrency can provide substantial benefits in performance, scalability, responsiveness, and system design when it is used appropriately.

2.4.1. Improved Performance

By dividing work into smaller independent units, an application can make better use of the available hardware resources [9, 10]. Any objective claim about performance gains from parallelization should be validated with appropriate benchmarks; in Java, the Java Microbenchmark Harness (JMH) is the standard tool for reliable microbenchmarking [41]. For example, an application that processes thousands of independent files or performs independent

calculations may distribute the work across several threads and reduce the total execution time.

Even on a single-core processor, concurrency can improve overall resource utilization when one task spends time waiting for I/O. While one thread is blocked waiting for data from the network or storage, another runnable thread can use the processor.

2.4.2. Scalability

Modern applications often need to cope with increasing data volumes and growing numbers of users [9, 10].

A web server handling concurrent requests, a parallel processing engine, or a distributed analysis application can all benefit from a concurrent design.

Thread-based and task-based abstractions allow work to be distributed dynamically across the available cores without requiring the application developer to manage processor assignment directly.

2.4.3. Responsiveness

One of the most visible benefits of concurrency is responsiveness: an application can remain available for new user actions or incoming requests while other work continues in the background.

In a graphical user interface, expensive operations such as downloading a file or querying a database can run away from the user-interface thread, allowing the interface to remain responsive instead of freezing.

Similarly, a concurrent server can continue accepting and processing requests from many clients instead of making every client wait for one long-running operation to complete.

2.4.4. A Natural Model for Many Real-World Problems

The world itself is concurrent: many activities take place at the same time and interact with one another.

For this reason, application domains such as simulations, games, industrial systems, video processing, and microservices often map naturally to multiple cooperating activities.

Concurrency is therefore not merely a performance optimization. It can also provide a useful conceptual model for structuring software around work that is naturally independent or overlapping.

In short, concurrent programming is a design strategy rather than an end in itself. Used carefully, it can make Java applications more responsive, scalable, and efficient; used without a clear model of synchronization and shared state, it can introduce difficult and nondeterministic bugs.

2.5. Challenges and Risks of Concurrent Programming

Concurrent programming provides important benefits, but it also introduces substantial complexity. When multiple threads interact with shared state, program behavior becomes harder to reason about and may depend on timing that changes from one execution to the next.

Even an apparently minor synchronization mistake can lead to failures that are intermittent and difficult to reproduce, making concurrent bugs especially challenging to diagnose.

The main hazards discussed in this section are race conditions, deadlocks, livelock, starvation, and memory-visibility problems.

2.5.1. Race Conditions

A race condition occurs when two or more threads access the same shared resource, such as a variable or object, without adequate coordination and the program result depends on the order in which their operations are interleaved. It is a logic error rather than a syntax error, which makes it particularly difficult to detect [1, 9, 14].

Consider the following simple example:

```
public class Counter {  
    private int value = 0;  
  
    public void increment() {  
        value++;  
    }  
  
    public int getValue() {  
        return value;  
    }  
}
```

If several threads call `increment()` concurrently, the final value may be incorrect. The expression `value++` is not an atomic operation: conceptually, it reads the current value, computes the incremented value, and writes the result back.

- read the current value;
- increment it;
- write the new value.

If two threads interleave these steps, one update can overwrite the other. Operations of this kind therefore require an appropriate synchronization mechanism, such as `synchronized`, `lock`, or `AtomicInteger`.

2.5.2. Deadlock

A deadlock occurs when two or more threads wait for one another in such a way that none of them can continue. The program may appear frozen even though the threads themselves still exist [1, 9, 39].

A classic example is:

- thread A holds the lock for object X and attempts to acquire the lock for object Y;
- thread B holds the lock for Y and attempts to acquire the lock for X;

- neither thread can proceed because each is waiting for a lock held by the other.

Deadlocks can be difficult to reproduce because they often depend on a particular timing and interleaving of operations. Avoiding them requires careful design, including consistent lock-ordering rules. Later chapters return to deadlocks and show how they can be investigated in practice.

2.5.3. Livelock

In a livelock, threads are not blocked in the usual sense. They remain active and continue reacting to one another, but their reactions prevent useful progress [1, 3, 8].

An intuitive analogy is two people trying to pass through a doorway at the same time. Each repeatedly steps aside to let the other pass, and both keep moving, yet neither actually gets through the doorway.

2.5.4. Thread Starvation

Starvation occurs when a thread is repeatedly denied processor time or access to a shared resource because other threads continually obtain the resources it needs [1, 3, 8]. It can result from unfair resource-acquisition patterns, lock contention, or scheduling behavior.

`Thread` priorities may influence scheduling on some platforms, but they do not provide a portable guarantee that a higher-priority Java thread will always run before a lower-priority one. Starvation should therefore be prevented through sound synchronization and resource-management design rather than by relying on priorities.

2.5.5. Visibility Problems

Even when threads do not deadlock, they can still encounter memory-visibility problems [1, 12, 14, 15].

When shared data is accessed without the synchronization required by the Java Memory Model (JMM), one thread is not necessarily guaranteed to observe another thread's updates in the way the programmer expects. Compiler optimizations, JVM transformations, and hardware memory behavior can all affect when reads and writes become observable across threads. The JMM defines the ordering and visibility guarantees on which correct concurrent Java programs rely.

To establish the required visibility relationships, Java provides mechanisms such as:

- `volatile`, for shared variables whose reads and writes require the visibility and ordering guarantees defined by the JMM;
- `synchronized`, which provides mutual exclusion and also establishes happens-before relationships between monitor release and subsequent acquisition.

Concurrency is powerful, but it comes with real risks. A poorly `synchronized` program can appear to work correctly hundreds or thousands of times and still fail under a different timing or workload.

Understanding race conditions, deadlocks, livelock, starvation, and visibility is therefore essential for writing robust concurrent software. The following chapters examine these problems and the mechanisms used to prevent or manage them in greater detail.

2.6. Concurrency Models

As Java has evolved, several models and programming styles have emerged for managing concurrent work. Each offers a different way to structure tasks, communication, and shared state. The appropriate choice depends on the application, the acceptable level of complexity, and the available hardware resources [2, 8, 9].

2.6.1. Thread-Based Model

The thread-based model is the traditional approach that has existed since the earliest versions of Java and underlies the rest of the language's concurrency infrastructure [1, 2, 3].

Work is expressed as threads that execute independently while potentially sharing memory and other process resources.

Advantages:

- direct control over thread creation, startup, interruption, and coordination;
- well suited to small examples and to learning the fundamentals;
- provides a direct way to understand synchronization, blocking, and inter-thread communication.

Limitations:

- manual thread management becomes difficult to scale in large applications;
- manual synchronization and error handling can make code complex and difficult to maintain;
- shared state creates a substantial risk of race conditions and deadlocks.

A simple example is:

```
new Thread(  
    () -> System.out.println(  
        "Executed in a separate thread")).start();
```

For readers who are less familiar with lambda expressions, the same example can be written with an anonymous class:

```
new Thread(  
    new Runnable() {  
        @Override  
        public void run() {  
            System.out.println("Executed in a separate thread");  
        }  
    })
```

```
    }  
    ).start();
```

`Thread` and `Runnable` are the fundamental class and interface used by this model. Direct thread management remains important for understanding Java concurrency, but application code often relies on higher-level abstractions such as executors and `CompletableFuture` instead of creating and coordinating platform threads manually.

2.6.2. Message-Passing Model

In the message-passing model, execution entities communicate by exchanging messages instead of directly sharing mutable memory. This design can eliminate many problems caused by uncontrolled concurrent access to shared state [8, 43].

A classic example is the Actor model, implemented by libraries and platforms such as Akka. Each actor owns a mailbox and processes incoming messages according to the framework's execution model. Other actors do not directly manipulate its internal state; interaction takes place through messages.

Advantages:

- reduces many synchronization problems associated with shared mutable state;
- supports scalable and distributed designs;
- maps naturally to reactive and cloud-native systems.

Limitations:

- communication between entities introduces overhead;
- message ordering, delivery, and failure handling add their own complexity;
- the Actor model is not part of the Java standard library and normally requires an external library or framework.

Because this model is not part of the standard Java concurrency APIs, it is not covered in detail in this book.

2.6.3. Asynchronous Model

The asynchronous model allows an operation to be started without requiring the caller to wait synchronously for its result. In Java, abstractions such as `Future` and `CompletableFuture` can represent results that will become available later [6, 34].

This style became especially prominent with Java 8 and is useful when independent operations need to be composed without tying the control flow to a simple sequence of blocking calls. It is frequently used around I/O-intensive work such as remote-service calls, database operations, and microservices.

Example:

```
CompletableFuture.supplyAsync(() -> getDataFromAPI())
    .thenApply(data -> process(data))
    .thenAccept(result -> System.out.println("Result: " +
        result));
```

Advantages:

- can use resources efficiently when operations spend time waiting;
- supports composition without blocking the calling thread at every step;
- enables a declarative pipeline of dependent actions.

Limitations:

- control flow can become harder to follow, especially when many stages are chained;
- debugging and exception handling are often more complex than in straightforward synchronous code;
- the exact execution order and execution context require careful understanding.

2.6.4. Virtual-Thread Model

Virtual threads, introduced as a permanent feature in Java 21, are one of the most important results of Project Loom and represent a major change in how Java can structure highly concurrent applications [44, 45].

A virtual thread is a lightweight `Thread` managed by the JVM. Unlike a traditional platform thread, it is not permanently mapped one-to-one to a `native` operating-system thread. This makes it practical to create very large numbers of threads for workloads dominated by blocking operations while retaining the familiar sequential programming style.

Example:

```
try (var executor =
    Executors.newVirtualThreadPerTaskExecutor()) {
    IntStream.range(0, 1000).forEach(i ->
        executor.submit(() -> {
            Thread.sleep(1000);
            return i;
        })
    );
}
```

Advantages:

- supports very large numbers of concurrent tasks, especially for blocking I/O workloads;
- keeps code close to the straightforward thread-per-task style instead of requiring callback-heavy asynchronous logic;
- integrates with existing Java APIs built around `Thread` and `ExecutorService`;
- simplifies the implementation of highly concurrent servers and clients that perform large amounts of blocking I/O.

Limitations:

- virtual threads are intended primarily for workloads with substantial blocking or waiting; they do not make CPU-bound work intrinsically faster;
- high concurrency can expose downstream limits such as database connections, rate limits, and other scarce resources;
- they do not remove the need for correct synchronization when mutable state is shared.

Java’s concurrency mechanisms have therefore evolved from direct control over platform threads toward increasingly higher-level abstractions. `Thread`-based programming remains the foundation; asynchronous APIs provide composition; and virtual threads bring the simplicity of straightforward synchronous code to applications that need very high levels of concurrency.

Since Java 21, developers can choose a concurrency model that combines a familiar programming style with a much larger number of concurrent blocking tasks. The important point is not that virtual threads replace every other mechanism, but that they substantially expand the set of problems that can be solved with simple, readable thread-per-task code.

2.7. Conclusion

Concurrency is a fundamental part of modern software. Its purpose is not simply to create more threads, but to coordinate independent activities efficiently and safely so that an application remains responsive, scalable, and correct.

This chapter established the conceptual foundation needed for the rest of the book: the distinction between concurrency and parallelism, the relationship between processes and threads, the main benefits and risks of concurrent execution, and the principal models used in modern Java.

FLORIN RADU

Continue Reading

This sample includes the complete first two chapters.

The complete book continues with:

- Lambda Expressions and the Stream API
- Threads in Java - The Traditional Model
- Modern Concurrency Mechanisms in `java.util.concurrent`
- Classic Problems in Concurrent Programming
- Design Patterns for Concurrent Programming
- Virtual Threads and Structured Concurrency (Project Loom)

The book also includes a companion GitHub repository with runnable chapter examples and additional real-world examples.

Get the complete book on Leanpub

<https://leanpub.com/concurrentprogramminginjava>

Concurrent Programming in Java

From Traditional Threads to Project Loom