ORACLE PRESS

Also Covers **Java SE 11** Developer Exam

OCP

ORACLE CERTIFIED PROFESSIONAL

JAVA SE 21 Developer — Exam 1Z0-830

JAVA SE 17 Developer — Exam 1Z0-829

Programmer's Guide

Khalid A. Mughal
Vasily A. Strelnikov

ORACLE

FREE SAMPLE CHAPTER |

Important Information About The Programmer's Guide

This Programmer's Guide is comprised of two parts:

- *Part I: OCP Java SE 21 Developer (Exam 1Z0-830)*
- *Part II: OCP Java SE 17 Developer (Exam 1Z0-829)*

Part I can be used in conjunction with Part II to prepare for the *OCP Java SE 21 Developer exam*.

Part II can be used to prepare for the *OCP Java SE 17 and Java 11 Developer exams*.

This page intentionally left blank

OCP

Oracle Certified Professional

Java SE 21Developer (Exam 1Z0-830)

Java SE 17 Developer (Exam 1Z0-829) Programmer's Guide

Khalid A. Mughal
Vasily A. Strelnikov

Pearson
Hoboken, New Jersey

Cover image: your/Shutterstock

Visit us on the Web: informit.com

main ISBN-13: 978-0-13-544598-3
main ISBN-10: 0-13-544598-1
epub ISBN-13: 978-0-13-544603-4
epub ISBN:10: 0-13-544603-1

Release 1, December 2024.

Part I:
OCP
Java SE 21 Developer
(Exam 1Z0-830)

This page intentionally left blank

To my life companion Marit Seljeflot,
and our precious daughters Nina Sulabha and Laila Meghana.
With all my love.

—K. A.M.

This page intentionally left blank

Part I: Contents

Part I: Figures

This page intentionally left blank

Part I: Tables

This page intentionally left blank

Part I: Examples

This page intentionally left blank

Preface to OCP Java SE 21 Developer (Exam 1Z0-830)

Part I: OCP Java SE 21 Developer of this book provides a comprehensive coverage of *new topics* that have come into Java 21 since the release of Java 17 and that are relevant for the *Java SE 21 Developer Professional Exam (1Z0-830)*. Primarily three new topics in Java 21 (*pattern matching for objects*, *sequenced collections*, and *virtual threads*) are included in the objectives for the new exam. Apart from that, the objectives for the *Java SE 21 Developer Professional Exam* did not change significantly from the objectives for the *Java SE 17 Developer Exam*.

Part II: OCP Java SE 17 Developer of this book provides extensive coverage for the following Java certifications:

- *Oracle Certified Professional: Java SE 17 Developer*, and its required *Java SE 17 Developer Exam (1Z0-829)*
- *Oracle Certified Professional: Java SE 11 Developer,* and its required *Java SE 11 Developer Exam (1Z0-819)*

Part I and Part II together provide a comprehensive resource for Java 21 certification:

- *Oracle Certified Professional: Java SE 21 Developer,* and its required *Java SE 21 Developer Professional Exam (1Z0-830)*

In this preface we provide the necessary details on how best to prepare and pass the *Java SE 21 Developer Professional Exam* using this programmer's guide. The syllabus for this exam is defined by a set of *exam objectives* that are presented in Appendix A in Part I. The *new* topics that have been added to the *Java SE 21 Developer Professional Exam* since Java 17 are covered in Part I, and Appendix A provides detailed references where the rest of the topics are covered in *Part II: OCP Java SE 17 Developer.*

About Part I: OCP Java SE 21 Developer

Our approach to writing Part I has not changed from the one we employed for our previous books, mainly because it has proved successful. The emphasis remains on analyzing code scenarios, rather than esoteric syntax of individual language constructs. The exam continues to require actual experience with the language, not just mere recitation of facts. We still claim that proficiency in the language is the key to success.

Part I is no different from our previous books in one other important aspect: It provides a mixture of theory and practice that enables readers to prepare for the exam.

All elements found in our previous books (e.g., examples, figures, tables, review questions, mock exam questions) can be found in this one as well. We continue to use UML (*Unified Modeling Language*) extensively to illustrate concepts and language constructs, and all numbered examples continue to be complete Java programs ready for experimentation.

Each topic is explained and discussed thoroughly with examples, and is backed by review questions to reinforce the concepts. Part I is not biased toward any particular platform, but provides platform-specific details where necessary.

Part I is primarily intended for professionals who want to prepare for the *Java SE 21 Developer Professional Exam*, but it is readily accessible to any programmer who wants to master the new topics. After mastering the language and the core APIs by working systematically through Part I and Part II, the reader can confidently sit for the exam.

Part I has a separate appendix (Appendix A) providing all the pertinent information on preparing for the exam. Since Part I only covers new topics that are relevant for the Java 21 exam, it should be used with Part II, which covers the remaining exam topics.

The table of contents; listings of tables, examples, and figures; and a comprehensive index facilitate locating topics discussed in Part I. Cross-references are provided where necessary to indicate the relationships between the various features of Java.

In particular, we draw attention to the following features of Part I:

Chapter Topics

Each chapter starts with a short summary of the topics covered in the chapter, pointing out the major concepts that are introduced.

Prerequisites

Each chapter starts with a short summary of topics that are prerequisites for the topics covered in the chapter. *Part II: OCP Java SE 17 Developer* readily provides coverage for these prerequisites.

Exam Objectives

Developer Exam Objectives

[0.1] Exam objectives that are covered in each chapter are stated clearly at the beginning of every chapter.

[0.2] The number in front of the objective identifies the exam objective, as defined by Oracle. The objectives are organized into major sections, detailing the curriculum for the exam.

[0.3] The objectives for the *Java SE 21 Developer Professional Exam* are reproduced verbatim in Appendix A. This appendix also maps each exam objective to relevant chapters and sections in Part I and in *Part II: OCP Java SE 17 Developer*.

Supplementary Topics

- Supplementary topics are Java topics that are *not* on the exam per se, but which the candidate is expected to know.
- Any supplementary topic is listed as a bullet at the beginning of the chapter.

Review Questions

Review questions are provided after every major topic to test and reinforce the material. The review questions predominantly reflect the kind of multiple-choice questions that can be asked on the actual exam. On the exam, the exact number of answers to choose for each question is explicitly stated. The review questions in Part I follow that practice.

Many questions on the actual exam contain code snippets with line numbers to indicate that complete implementation is not provided, and that the necessary missing code to compile and run the code snippets can be assumed. The review questions in Part I provide complete code implementations where possible, so that the code can be readily compiled and run.

Annotated answers to the review questions are provided in Appendix B in Part I.

Example 0.1 *Example Source Code*

We encourage readers to experiment with the code examples to reinforce the material in the book. The source code for the examples can be downloaded from the website for Part I (see p. xviii), and readily imported into the Eclipse IDE.

Java code in the book is presented in a `monospaced font`. Lines of code in the examples or in code snippets are referenced in the text by a number, which is specified by using a single-line comment in the code. For example, in the following code snippet, the call to the method `doSomethingInteresting()` at (1) does something interesting:

```
// ...
doSomethingInteresting();                          // (1)
// ...
```

Names of classes and interfaces start with an uppercase letter. Names of packages, variables, and methods start with a lowercase letter. Constants are in all uppercase letters. Interface names begin with the prefix `I`, when it makes sense to distinguish them from class names. Coding conventions are followed, except when we have had to deviate from these conventions in the interest of space or clarity.

Java SE Platform API Documentation

`fully.qualified.Name`

A vertical gray bar is used to highlight methods and fields found in the classes of the Java SE Platform API.

Any explanation following the API information is also similarly highlighted.

To get the maximum benefit from using Part I in preparing for the *Java SE 21 Developer Professional Exam*, we strongly recommend installing the latest version (Release 21 or newer) of the JDK and its accompanying API documentation. Part I focuses solely on the new topics that were finalized in Java SE 21 since the release of Java SE 17.

Website for Part I: OCP Java SE 21 Developer

Part I is backed by a website:

`https://www.mughal.no/jse21ocp/`

Auxiliary material on the website includes the following:

- Source code for all the examples in Part I
- Annotated answers to the reviews questions in Part I
- Table of contents, sample chapters, and index from Part I
- Content specific for the *Java SE 21 Developer Professional Exam*
- Errata for Part I
- Links to miscellaneous Java resources (e.g., certification, discussion groups, and tools)

Information about the Java Standard Edition (SE) and its documentation can be found at the following website:

```
www.oracle.com/technetwork/java/javase/overview/index.html
```

The current authoritative technical reference for the Java programming language, *The Java® Language Specification: Java SE 21 Edition*, can be found at this website:

```
http://docs.oracle.com/javase/specs/index.html
```

Request for Feedback

Considerable effort has been made to ensure the accuracy of the content of this book. All code examples (including code fragments) have been compiled and tested on various platforms. In the final analysis, any errors remaining are the sole responsibility of the principal author.

Any questions, comments, suggestions, and corrections are welcome. Let us know whether the book was helpful (or not) for your purpose. Any feedback is valuable. The principal author and the co-author can be reached at the following e-mail addresses, respectively:

```
khalid@mughal.no
vasiliy.a.strelnikov@oracle.com
```

About the Authors

Khalid A. Mughal

Khalid A. Mughal is the principal author of this book, primarily responsible for writing the material covering the Java topics. He is also the principal author of several other books on previous versions of the Java certification exam: Java SE 17 OCP (1Z0-829), Java SE 11 OCP (1Z0-819), Java SE 8 OCA (1Z0-808), Java SE 6 (1Z0-851), and SCPJ2 1.4 (CX-310-035).

Khalid is an associate professor emeritus at the Department of Informatics at the University of Bergen, Norway, where he was responsible for designing and implementing various courses in informatics. Over the years, he has taught programming (primarily Java), software engineering (object-oriented system development), databases (data modeling and database management systems), compiler techniques, web application development, and software security courses. For 15 years, he was responsible for developing and running web-based programming courses in Java, which were offered to off-campus students. He has also given numerous courses and seminars at various levels in object-oriented programming and system development using Java and Java-related technologies, both at the University of Bergen and various other universities in Norway and East Africa, and also for the IT industry.

Vasily A. Strelnikov

Vasily Strelnikov is primarily responsible for developing new review questions for the chapters contained in this book.

Vasily is a senior principal OCI (Oracle Cloud Infrastructure) solution specialist, working at Oracle for more than 27 years. He is a co-author of the Java EE 7, Java SE 8, Java SE 11, Java SE 17, and Java SE 21 Certification exams. He has designed multiple Java courses that are offered by Oracle: Java SE 21 Developer, Java SE 17 Programming Complete, Java SE 11 Programming Complete, and Java SE 8 Certification Preparation Seminar. He has also created the Developing Applications for the Java EE 7 Platform training at Oracle. Vasily has over 25 years of experience in Java. He specializes in Java middleware application development and web services.

Acknowledgments

At Pearson, senior executive editor Greg Doench was once again in charge of this project. Senior content producer Julie Nahil was again the in-house contact at Pearson, managing the production of the book professionally and efficiently. Freelancer Deborah Woods did a meticulous job copyediting Part I of this book. Our sincere thanks to Greg, Julie, Deborah, and all those behind the scenes, who helped to get this publication out the door.

For the technical review of Part I, we were again lucky to have a Java guru who graciously agreed to take on the task:

> Mikalai Zaikin is a lead Java developer at IBA Lithuania, and is currently located in Vilnius. He has helped Oracle with development of Java certification exams and has also been a technical reviewer of several Java certification books. He also contributes to the Java Quiz column for Oracle's *Java Magazine* in collaboration with Simon Roberts.
>
> Without doubt, Mikalai has an eye for detail. It is no exaggeration to say that his feedback has been invaluable in improving the quality of Part I at all levels. Our most sincere thanks to Mikalai for the many excellent comments and suggestions on the contents—especially regarding code examples and review questions, and above all, for weeding out numerous pesky errors in the manuscript.

Great effort has been made to eliminate mistakes and errors in this book. Any remaining oversights are solely our responsibility. We hope that our readers will bring them to our attention when they find them.

Family support was undoubtedly essential in this project as well and for that we are very grateful to our families for putting up with us.

We wish our reader all the best in going down the caffeine-infused path of Java certification. May your threads stay untangled and complete normally!

—Khalid A. Mughal
November 5, 2024
Bergen, Norway

This page intentionally left blank

Virtual Threads 3

Chapter Topics

- What virtual threads are and how they compare to platform threads.
- How virtual threads are executed.
- Creating and using virtual threads.
- Using virtual thread executors to run tasks.
- Best practices for using virtual threads.

Prerequisites

- Understanding how platform threads are executed.
- Understanding the thread lifecycle.
- Creating and using platform threads.
- Using executor services to run tasks.

Java SE 21 Developer Professional Exam Objectives		
[8.1]	Create both platform and virtual threads. Use both Runnable and Callable objects, manage the thread lifecycle, and use different Executor services and concurrent API to run tasks.	*Only virtual threads are covered in this chapter.*

Introduction of *virtual threads* promises to facilitate building very high-throughput concurrent applications that employ the one-thread-per-task paradigm. These lightweight threads are under the regime of the JVM, and not the operating system. In this chapter we cover what virtual threads are, explain their high-throughput execution model, create and use them to run tasks, compare them to platform threads, and provide best practices for using them. Most importantly, we need hardly learn a new API to utilize them.

Before going forward, it is highly recommended to have a sound understanding of the traditional concurrency model based on platform threads as outlined in the prerequisites at the start of this chapter.

3.1 Motivation for Virtual Threads

The concurrency model in Java has traditionally centered around *platform threads*—threads that are scheduled by the operating system and mapped to *operating system (OS) threads* in order for them to execute Java code.

Concurrent applications based on *one-thread-per-task paradigm* strive to increase their *throughput*—that is, increase the number of tasks that can be done concurrently—by requiring evermore platform threads. However, constraints on the number of platform threads that can be created and managed is often the bottleneck when it comes to scalability in concurrent applications. The constraints can be due to limitations imposed by the hardware, the operating system, or the sheer size of memory that would be required to handle vast number of platform threads. One-thread-per-task style of developing high-throughput concurrent applications with platform threads therefore becomes infeasible.

3.2 Virtual Thread Execution Model

By design, virtual threads are *lightweight*:

- They are less expensive to create and destroy than platform threads.
- They require substantially less *stack memory* to manage their execution than platform threads.
- *Context switching* between virtual threads is far less expensive than between platform threads.

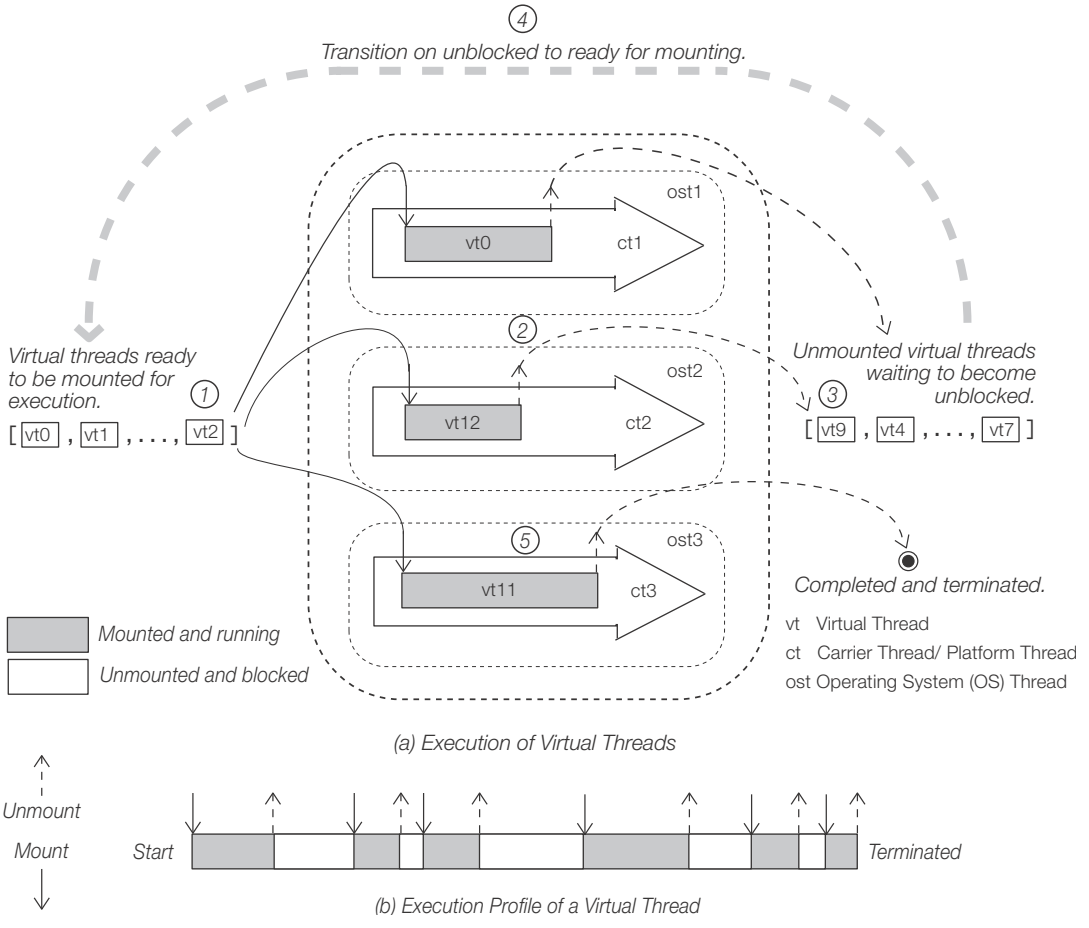
Virtual threads are managed by the JVM scheduler during their lifetime, as opposed to platform threads that have a one-to-one mapping with OS threads and are scheduled by the operating system during their lifetime. In other words, the whole regime of virtual threads is managed by the JVM that has a pool of platform threads at its discretion, without the intervention of the operating system. Because of their nature, virtual threads are ideal for building concurrent applications based

on the one-thread-per-task paradigm, allowing each task to execute in its own virtual thread.

Execution of virtual threads is illustrated in Figure 3.1a. As with platform threads, a virtual thread is first created to run a task and then scheduled to begin execution. There can be many virtual threads scheduled to begin execution ((1) in Figure 3.1a). From hereon their execution is at the discretion of the JVM.

In order to run the task in a virtual thread that is ready for execution, the JVM scheduler assigns the virtual thread to a *platform thread* for execution—this is called *mounting* the virtual thread (`vt`) and the designated platform thread is called the *carrier thread* (`ct`) ((2) and (5) in Figure 3.1a showing virtual threads `vt0`, `vt12`, and `vt11` that are mounted on carrier threads `ct1`, `ct2`, and `ct3`, respectively). A platform thread is mapped to an OS thread (`ost`) and acts as a carrier thread when it is executing a virtual thread.

Figure 3.1 *Virtual Thread Execution Model*



Executing certain operations in its task can cause a mounted virtual thread to *unmount* from its carrier thread—that it, to relinquish its carrier thread (as at (2) in Figure 3.1a for virtual threads `vt0` and `vt12`). A virtual thread is unmounted when it executes a *blocking operation* (such as an I/O operation). It does not require any action on the part of the application to initiate unmounting when a blocking operation is executed. I/O operations and other relevant blocking operations in the APIs have been updated to work with virtual threads. The unmounted virtual thread remains blocked until its blocking operation is ready to complete ((3) in Figure 3.1a), at which point, it is unblocked and joins other virtual threads waiting to mount and thus resume execution ((4) in Figure 3.1a).

A virtual thread that completes its execution while mounted is of course unmounted and terminated ((5) in Figure 3.1a for virtual thread `vt11`) and its carrier thread used to mount another virtual thread that is ready to be mounted for execution.

Instead of a carrier thread being monopolized by its virtual thread until the blocked operation is ready to complete, unmounting it allows the JVM scheduler to mount another virtual thread that is ready for execution on the carrier thread.

An execution profile of a virtual thread is shown in Figure 3.1b, illustrating that the virtual thread executes when it is mounted on a carrier thread, and is unmounted and blocked on a blocking operation until the operation is ready to complete. Note that on resumption of its execution, a virtual thread may be mounted on the same carrier thread or on a different carrier thread.

The ratio `m:n` of `m` virtual threads to `n` platform threads is usually very high, contributing to the high-throughput of the virtual thread execution model. A task assigned to a platform thread remains assigned to the platform thread throughout its lifetime—even while it is in a waiting or a blocked state, and cannot therefore do any useful work. The task in a virtual thread also remains assigned to the same virtual thread through its lifetime, but the virtual thread may be executed on one or more platform threads, freeing the carrier thread to do other work when the virtual thread is unmounted and blocked. The virtual thread execution model results in high utilization of the platform threads by multitude of virtual threads, which counts for the high throughput of the one-thread-per-task execution model.

Platform threads are also known as *classical threads* or *traditional threads*. *OS threads* are also known as *native threads* or *kernel threads*.

3.3 Using `Thread` Class to Create Virtual Threads

The `Thread` class supports virtual threads and provides new methods for this purpose. We cannot use a constructor of the `Thread` class to create a virtual thread as all constructors create platform threads. However, the Concurrency API provides great flexibility in creating and running virtual threads, as we will see in the rest of this chapter.

Logging Information during Program Execution

Using *blocking* operations (such as print methods of the `System.out` stream) to write information about program execution in concurrent applications can adversely affect program performance. The Logging API provides a simple, yet flexible *non-blocking* mechanism to log such information. There are no direct questions on the exam in this topic, but use of loggers may be encountered in the context of other questions, such as in the context of the Concurrency API or execution of parallel streams.

Steps (1)-(4) shown in Example 3.1 are sufficient to create and use a logger for our purpose, which we will be doing in some examples in this chapter:

(1) Import the `Logger` class.

(2) Declare a `static` field and create an instance of the `Logger` class.

(3) Provide a static initializer block to set property for appropriate format to be used by the formatter when logging messages. For example:
`[Timestamp] INFO: ...`

(4) Call the `info()` method of the logger in the program to log messages.

Example 3.1 *Using a Logger*

```
package vt;
import java.util.logging.Logger;                                  // (1)

public class Main {
  private static final Logger logger =                            // (2)
      Logger.getLogger(Main.class.getName());

  static {                                                        // (3)
    System.setProperty("java.util.logging.SimpleFormatter.format",
                       "[%1$tT.%1$tN] %4$s: %5$s%n");
  }

  public static void main(String[] args) {
    logger.info("Log this info.");                                // (4)
  }
}
```

Probable output from the program:

```
[12:22:11.357397000] INFO: Log this info.
```

Creating and Starting a Virtual Thread

The simplest way to *create and start* a virtual thread is to use the `static` method `startVirtualThread()` of the `Thread` class, as shown at (2) in Example 3.2, passing the task that is to be executed.

```
Thread vt = Thread.startVirtualThread(task);                    // (2)
```

A task is defined at (1) as a `Runnable` that prints the *string representation* of the current thread, which in this case will be a virtual thread, when the task is executed. Note that the print method is a *blocking* operation.

```
Runnable task = () -> System.out.println(Thread.currentThread());     // (1)
```

The virtual method created and started at (2) will execute the task at (1) when it is allowed to run, printing information about the virtual thread:

```
VirtualThread[#21]/runnable@ForkJoinPool-1-worker-1
```

In this case, the string representation of the current thread comprises of the following components:

- `VirtualThread` identifies that it is a virtual thread.
- `#21` specifies the unique *thread ID* of the virtual thread. In this case it is `21`. The thread ID does not change during the lifetime of a thread.
- `runnable` indicates the *state* the thread is in. In this case, it is in the *runnable state*.
- `ForkJoinPool-1-worker-1` is composed of the *name of the fork-join pool* and the *name of the carrier thread* on which the virtual thread is mounted. The name `ForkJoinPool-1` identifies the fork-join pool that manages the carrier threads (which are platform threads). Designation `worker-1` is the name of the carrier thread in the fork-join pool `ForkJoinPool-1` on which the virtual thread with ID `#21` is mounted.

  The number value in the thread ID designation, the carrier thread designation, and the fork-join pool designation are counter values giving an indication of how many of these entities have been created.

Virtual threads are daemon threads—that is, they are unceremoniously terminated when the parent platform thread terminates. The virtual thread created at (2) in Example 3.2 can risk being terminated before it has completed if the parent platform thread (in this case, the *main* thread) completes first. In order to allow the virtual thread to complete its execution, the parent thread can call the `join()` method on the virtual thread in order to wait for its completion before proceeding. The call at (4) will ensure that the main thread will wait indefinitely and does not proceed before the virtual thread completes its execution.

```
vt.join();                                                      // (4)
```

In Example 3.2, the main thread logs information about whether the virtual thread is alive before and after calling the `join()` method on the virtual thread. The logged information shows that virtual thread `#21` was alive before the call to the `join()` method, but had completed its execution after the call.

Finally, the information logged at (5) in Example 3.2 shows that a virtual thread is an instance of the `java.lang.VirtualThread` class. This class is a *non-public subclass* of the `Thread` class in the `java.lang` package and not accessible outside this package. For all intents and purposes, it is the `Thread` class that provides the support for vir-

tual threads. However, the application has to keep track of whether a reference of type Thread denotes a virtual or a platform thread.

Example 3.2 *Creating and Running a Virtual Thread*

```
package vt;
import java.util.logging.Logger;

public class BasicVTCreation {

  private static final Logger logger =
      Logger.getLogger(BasicVTCreation.class.getName());
  static {
    System.setProperty("java.util.logging.SimpleFormatter.format",
        "[%1$tT.%1$tN] %4$s: %5$s%n");
  }

  public static void main(String[] args) throws InterruptedException {

    // Create a task:
    Runnable task = () -> System.out.println(Thread.currentThread());     // (1)

    // Create and start a virtual thread that is assigned a task:
    Thread vt = Thread.startVirtualThread(task);                          // (2)

    logger.info("Before join, vt #" + vt.threadId() +                     // (3)
        " is " + (vt.isAlive() ? "alive." : "not alive."));

    vt.join();                                                            // (4)

    logger.info("After join, vt #" + vt.threadId() +                      // (5)
        " is " + (vt.isAlive() ? "alive." : "not alive."));

    // Class of a virtual thread:
    logger.info("A virtual thread is an instance of " + vt.getClass());   // (6)
  }
}
```

Probable output from the program:

```
VirtualThread[#21]/runnable@ForkJoinPool-1-worker-1
[17:43:54.118896000] INFO: Before join, vt #21 is alive.
[17:43:54.185280000] INFO: After join, vt #21 is not alive.
[17:43:54.186162000] INFO: A virtual thread is an instance of class java.lang.Vir-
tualThread
```

The following is a summary of selected new methods in the `java.lang.Thread` class since Java 17:

`java.lang.Thread`

`static Thread startVirtualThread(Runnable task)`

Creates a virtual thread to execute a task and schedules it for execution.

This method is equivalent to: `Thread.ofVirtual().start(task);`

`static Thread.Builder.OfVirtual ofVirtual()`

Returns a *builder* for creating a virtual `Thread` or `ThreadFactory` that can be used to create virtual threads.

`static Thread.Builder.OfPlatform ofPlatform()`

Returns a *builder* for creating a platform `Thread` or `ThreadFactory` that can be used to create platform threads.

`final boolean isVirtual()`

Returns `true` if this thread is a virtual thread. A virtual thread is scheduled by the JVM rather than the operating system.

`final long threadId()`

Returns the identifier of this `Thread`. The thread ID is a positive `long` number generated when this thread was created. The thread ID is unique and remains unchanged during its lifetime.

`final boolean join(Duration duration) throws InterruptedException`

A call to this overloaded method invoked on a thread will wait and not return until either the thread has completed or it is timed out after the specified duration. It returns `true` if the thread has terminated, `false` if the thread has not terminated.

It does not wait if the duration to wait is less than or equal to zero. In this case, it just tests if the thread has terminated.

`static void sleep(Duration duration) throws InterruptedException`

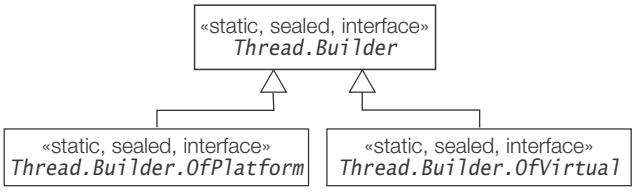
The current thread sleeps (that is, temporarily ceases execution) for at least the specified time before it becomes eligible for running again. Note this is a *blocking* operation on the current thread.

3.4 Using Thread Builders to Create Virtual Threads

For greater flexibility in creating threads, the `Thread` class defines *nested interfaces* that are *builders* for creating *threads* and *thread factories* (p. 86). In addition, a thread builder provides methods to set thread properties like the thread name, which once set, are valid for all threads and thread factories created with the thread

builder. The inheritance hierarchy of these nested interfaces defined in the `Thread` class is shown in Figure 3.2, where the thread builder subinterfaces `Thread.Builder.OfPlatform` and `Thread.Builder.OfVirtual` pertain to platform and virtual threads, respectively.

Figure 3.2 *Inheritance Hierarchy of Thread Builders*



An implementation of a virtual thread builder (that implements the `Thread.Builder.OfVirtual` interface) or a platform thread builder (that implements the `Thread.Builder.OfPlatform` interface) is obtained by calling the static methods `ofVirtual()` or `ofPlatform()` of the `Thread` class, respectively.

Example 3.3 demonstrates using thread builders to create and start threads. The virtual thread builder returned by the `Thread.ofVirtual()` method has the name property set by the `Thread.Builder.OfVirtual.name()` method. The threads it creates will have the name `"VT_n"`, where the prefix `"VT_"` is concatenated with the string representation of `n` that is the value of the counter that the thread builder employs, starting with the initial value specified together with the prefix in the call to the `name()` method.

```
Thread.Builder.OfVirtual vtBuilder = Thread.ofVirtual().name("VT_", 1);
```

The assignment statement above is equivalent to the following statements:

```
Thread.Builder.OfVirtual vtBuilder = Thread.ofVirtual();
vtBuilder.name("VT_", 1);// Returned reference to the thread builder is discarded.
```

The printout shows that the two virtual threads created have the names `VT_1` and `VT_2` in their default string representation.

Calling the `name()` method with only the string name will set the same name for all threads created by the thread builder. Note that a virtual thread does not have a name when it is created.

Calling the `start()` or the `unstarted()` methods of the thread builder will create a thread when passed a `Runnable`—that is, the task to execute, but only the thread created by the `start()` method of the thread builder will be scheduled to begin execution, and the thread created by the `unstarted()` method of the thread builder must explicitly call its `start()` method to begin execution. The following code is equivalent to the code at (3):

```
Thread vt1 = vtBuilder.unstarted(task);
Thread vt2 = vtBuilder.unstarted(task);
vt1.start();
```

```
vt2.start();
```

Printout from Example 3.3 shows that VT_1 and VT_2 were mounted on carrier threads worker-1 and worker-2 during execution of the task.

When the code at (4) is executed to print the string representation of the virtual threads, we see that thread VT_1 is still in the runnable state but not mounted on any carrier thread. However, thread VT_2 is in the terminated state having completed its execution.

The join() method is necessary to call at (5) to allow the virtual threads to complete their execution.

Creation of platform threads using a platform thread builder is analogous to that for virtual threads, as shown from (6) to (8) in Example 3.3. Waiting to join in the main thread is not necessary for platform threads. The string representation of a platform thread includes the thread ID, the thread name if any, the priority, and the name of the parent thread. The Thread.Builder.ofPlatform interface defines methods to set various properties of a platform thread.

Example 3.3 *Using Thread Builders to Create Threads*

```
package vt;
public class ThreadBuilderDemo {
  public static void main(String[] args) throws InterruptedException {

    // Create a task:                                                    (1)
    Runnable task = () -> System.out.printf("%s: I am on it!%n",
                                          Thread.currentThread());

    // Obtain a virtual thread builder:                                  (2)
    Thread.Builder.OfVirtual vtBuilder = Thread.ofVirtual().name("VT_", 1);

    // Creating and starting 2 virtual threads                           (3)
    // using the virtual thread builder:
    Thread vt1 = vtBuilder.start(task);
    Thread vt2 = vtBuilder.start(task);

    // Print virtual thread info:                                        (4)
    System.out.println("vt1: " + vt1);
    System.out.println("vt2: " + vt2);

    // Wait for the virtual threads to join:                             (5)
    vt1.join();
    vt2.join();

    System.out.println(new StringBuffer().repeat('-', 40));

    // Obtain a platform thread builder:                                 (6)
    Thread.Builder.OfPlatform ptBuilder = Thread.ofPlatform().name("PT_", 1);

    // Creating and starting 2 platform threads                          (7)
    // using the platform thread builder:
```

```
        Thread pt1 = ptBuilder.start(task);
        Thread pt2 = ptBuilder.start(task);

        // Print platform thread info:                                    (8)
        System.out.println("pt1: " + pt1);
        System.out.println("pt2: " + pt2);
    }
}
```

Probable output from the program:

```
VirtualThread[#22,VT_2]/runnable@ForkJoinPool-1-worker-2: I am on it!
VirtualThread[#20,VT_1]/runnable@ForkJoinPool-1-worker-1: I am on it!
vt1: VirtualThread[#20,VT_1]/runnable
vt2: VirtualThread[#22,VT_2]/terminated
----------------------------------------
Thread[#25,PT_1,5,main]: I am on it!
pt1: Thread[#25,PT_1,5,main]
Thread[#26,PT_2,5,main]: I am on it!
pt2: Thread[#26,PT_2,5,main]
```

java.lang.Thread.Builder

`Thread unstarted(Runnable task)`

Creates a new `Thread` from the current state of the builder to run the given task but does *not* schedule it to execute.

`Thread start(Runnable task)`

Creates a new `Thread` from the current state of the builder and schedules it to execute.

`Thread.Builder name(String name)`

The thread name that will be used for any thread created by this thread builder.

`Thread.Builder name(String prefix, long start)`

The thread name will be the concatenation of a string prefix and the string representation of a counter value for any thread created using this thread builder.

`ThreadFactory factory()`

Returns a `ThreadFactory` to create threads from the current state of the builder.

java.lang.Thread.Builder.OfVirtual extends java.lang.Thread.Builder

`Thread.Builder.OfVirtual name(String name)`

The thread name that will be used for any virtual thread created by this virtual thread builder.

java.lang.Thread.Builder.OfVirtual extends java.lang.Thread.Builder (Continued)

`Thread.Builder.OfVirtual name(String prefix, long start)`

The thread name will be the concatenation of a string prefix and the string representation of a counter value for any virtual thread created using this virtual thread builder.

java.lang.Thread.Builder.OfPlatform extends java.lang.Thread.Builder

`Thread.Builder.OfPlatform name(String name)`

The thread name that will be used for any platform thread created by this platform thread builder.

`Thread.Builder.OfPlatform name(String prefix, long start)`

The thread name will be the concatenation of a string prefix and the string representation of a counter value for any platform thread created using this platform thread builder.

`default Thread.Builder.OfPlatform daemon()`

Sets the daemon status to `true` for any platform thread created by this platform thread builder.

`Thread.Builder.OfPlatform daemon(boolean on)`

Sets the daemon status for any platform thread created by this platform thread builder.

`Thread.Builder.OfPlatform group(ThreadGroup group)`

Sets the thread group of any platform thread created by this platform thread builder.

`Thread.Builder.OfPlatform priority(int priority)`

Sets the thread priority of any platform thread created by this platform thread builder.

`Thread.Builder.OfPlatform stackSize(long stackSize)`

Sets the desired stack size for any platform thread created by this platform thread builder.

Important Aspects of Virtual Threads

Example 3.4 shows how virtual threads and platform threads are different in various aspects. Two unstarted threads, one virtual thread and one platform thread, are created with names `vt` and `pt` using a virtual and a platform thread builder, respectively.

Is a Thread Virtual?

The `isVirtual()` method of the `Thread` class determines whether a thread is virtual or not. The output shows that thread `vt` is virtual but thread `pt` is not. There is no `isPlatform()` method in the `Thread` class.

Virtual Threads are Daemon Threads

The `isDaemon()` method of the `Thread` class determines whether a thread is daemon or not. The output shows that thread `vt` is daemon but thread `pt` is not. Trying to set a virtual thread as non-daemon with the `setDaemon(false)` call results in an `IllegalArgumentException`.

Virtual Threads have Normal Priority

Virtual threads always have `ThreadPriority.NORM_PRIORITY` (=5) that cannot be changed. Setting a different priority of a virtual thread with the `setPriority()` method is ignored. The method can readily be used to change the priority of a platform thread.

Virtual Threads belong to VirtualThreads Group

All virtual threads belong to the *VirtualThreads* group, whereas a platform thread belongs in a specific thread group. A thread group allows its thread to be manipulated collectively rather than individually. The output shows that thread `vt` belongs to the *VirtualThreads* group, whereas thread `pt` belongs to the *main* thread group.

Example 3.4 *Selected Aspects of Threads*

```
package vt;

public class ThreadAspects {
  public static void main(String[] args) throws InterruptedException {

    // Create task:
    Runnable task = () -> System.out.println(Thread.currentThread());

    // Create threads:
    Thread vt = Thread.ofVirtual().name("vt").unstarted(task);
    Thread pt = Thread.ofPlatform().name("pt").unstarted(task);

    // Get names:
    String vtName = vt.getName();
    String ptName = pt.getName();

    // Virtual:
    System.out.println(vtName + " virtual? " + vt.isVirtual());
    System.out.println(ptName + " virtual? " + pt.isVirtual());

    // Daemon:
    System.out.println(vtName + " daemon? " + vt.isDaemon());
```

```
        // vt.setDaemon(false);              // java.lang.IllegalArgumentException:
                                             // can only be true for virtual threads
        System.out.println(ptName + " daemon? " + pt.isDaemon());

        // Priority:
        System.out.println(vtName + " priority (before change): " +
                           vt.getPriority()); // NORM_PRIORITY = 5
        vt.setPriority(6);
        System.out.println(vtName + " priority (after change): " +
                           vt.getPriority()); // Unchanged: NORM_PRIORITY

        System.out.println(ptName + " priority (before change): " +
                           pt.getPriority()); // NORM_PRIORITY = 5
        pt.setPriority(4);
        System.out.println(ptName + " priority (after change): " + pt.getPriority());

        // ThreadGroup:
        System.out.println("Thread group for " + vtName + ": "  +
                           vt.getThreadGroup().getName());
        System.out.println("Thread group for " + ptName + ": "  +
                           pt.getThreadGroup().getName());

        vt.start();
        vt.join();

        pt.start();
      }
    }
```

Probable output from the program:

```
vt virtual? true
pt virtual? false
vt daemon? true
pt daemon? false
vt priority (before change): 5
vt priority (after change): 5
pt priority (before change): 5
pt priority (after change): 4
Thread group for vt: VirtualThreads
Thread group for pt: main
VirtualThread[#20,vt]/runnable@ForkJoinPool-1-worker-1
Thread[#21,pt,4,main]
```

3.5 Using Thread Factory to Create Threads

A *thread factory* can be used to create threads on demand. Such a factory implements the `newThread()` method of the `ThreadFactory` interface that creates and returns an unstarted thread. A thread factory is *thread-safe* from multiple concurrent threads, as opposed to a thread builder which is not.

Thread builders provide the `factory()` method that returns a thread factory based on the current state of the builder, such as the thread name to use when creating threads.

```
ThreadFactory vtf = Thread.ofVirtual().name("VT_", 1).factory(); // (2)
```

In the code above from Example 3.5, the *virtual thread factory* (`ThreadFactory`) obtained from the virtual thread builder (`Thread.Builder.OfVirtual`) that is returned by the `Thread.ofVirtual()` method will create unstarted virtual threads whose names will be `VT_1`, `VT_2`, and so on.

Unstarted virtual threads are created at (2) by calling the `newThread()` method of the thread factory and have to be explicitly scheduled to begin execution by calling the `start()` method of the `Thread` class:

```
Thread vt4 = vtf.newThread(task);       // VT_1
...
vt4.start();
```

Using a platform thread factory obtained from a platform thread builder is analogous to using a virtual thread factory obtained from a virtual thread builder.

Example 3.5 *Using a Virtual Thread Factory to Create Virtual Threads*

```
package vt;
import java.util.concurrent.ThreadFactory;

public class ThreadFactoryDemo {
  public static void main(String[] args) throws InterruptedException {

    // Create a task:                                                    (1)
    Runnable task = () -> System.out.printf("%s: I am on it!%n",
                                            Thread.currentThread());

    // Obtain a virtual thread factory using a virtual thread builder:
    ThreadFactory vtf = Thread.ofVirtual().name("VT_", 1).factory(); // (2)

    // Create virtual threads using the virtual thread factory          (3)
    Thread vt4 = vtf.newThread(task);       // VT_1
    Thread vt5 = vtf.newThread(task);       // VT_2

    vt4.start();
    vt5.start();

    vt4.join();
    vt5.join();
  }
}
```

Probable output from the program:

```
VirtualThread[#21,vt_5]/runnable@ForkJoinPool-1-worker-2: I am on it!
```

```
VirtualThread[#20,vt_4]/runnable@ForkJoinPool-1-worker-1: I am on it!
```

java.util.concurrent.ThreadFactory

`Thread newThread(Runnable r)`

Constructs a new unstarted `Thread` to run the given runnable.

Returns the constructed thread, or `null` if the request to create a thread is rejected.

3.6 Using Thread Executor Services

An *executor service* implements the `ExecutorService` interface that extends the `Executor` interface. It provides methods that facilitate:

- *Flexible submitting of tasks* to the executor service and *handling of results* that are returned from executing tasks.
- *Managing the lifecycle of an executor service*: creating, running, shutdown, and termination of the executor service.

The `Executors` utility class provides two methods `newVirtualThreadPerTaskExecutor()` and `newThreadPerTaskExecutor()` to obtain one-thread-per-task executor services. Both the executor services implement the `AutoCloseable` interface and are thus best deployed in a `try`-with-resources construct that ensures proper shutdown and termination of the executor service.

Using the Virtual-Thread-Per-Task Executor Service

The method `newVirtualThreadPerTaskExecutor()` returns an executor service that embodies the *one-virtual-thread-per-task* model of execution—in other words, it exclusively uses a new virtual thread to execute a task.

In Example 3.6, the code at (2) creates a *one-virtual-thread-per-task executor service* that will create a virtual thread for each task that is submitted. Note there is no way to set any property of a virtual thread that the executor service creates. For example, we cannot set the name of a virtual thread in this executor service.

Customizing the Thread-Per-Task Executor Service

The method `newThreadPerTaskExecutor()` returns an executor service that is more customizable by a thread factory for executing one thread per task in general. The kind of threads the executor service will create and deploy depends on the thread factory passed to the method.

In Example 3.6, the code at (3) creates a *one-thread-per-task executor service* that will create a new virtual thread for each task that is submitted, as it is passed a *virtual thread factory* that is also customized to use a naming scheme for the virtual threads created. This naming scheme for the virtual threads is reflected in the output.

Note that submitting a task to an executor service using the `submit()` method is an *asynchronous* operation—that is, the method returns immediately. The `try`-with-resources construct used to manage the executor service ensures that there is an orderly shutdown and termination of the executor service when the submitted tasks have completed execution.

The handling of the result returned by a task that is implemented as a `Callable<V>` object and submitted to an execution service for execution by a virtual thread is no different than if it was by a platform thread, requiring polling of the `Future<V>` object that receives the result.

Example 3.6 *Using One-Thread-Per-Task Executor Service*

```
package vt;
import java.util.concurrent.*;
import java.util.stream.IntStream;

public class OneThreadPerTaskExecutorDemo {
  public static final int NUM_OF_TASKS = 5;

  public static void main(String[] args) throws InterruptedException {

    // Create a task:                                                    (1)
    Runnable task = () -> System.out.printf("%s: I am on it!%n",
        Thread.currentThread());

    // Using an ExecutorService for running one virtual thread per task:   (2)
    try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
      IntStream.range(0, NUM_OF_TASKS).forEach(i -> executor.submit(task));
    }

    System.out.println(new StringBuffer().repeat('-', 69));

    // Using a customized virtual thread factory with an ExecutorService   (3)
    // for running one virtual thread per task.
    ThreadFactory vtf = Thread.ofVirtual().name("VT_", 1).factory();
    try (ExecutorService executor = Executors.newThreadPerTaskExecutor(vtf)) {
      IntStream.range(0, NUM_OF_TASKS).forEach(i -> executor.submit(task));
    }
  }
}
```

Probable output from the program:

```
VirtualThread[#22]/runnable@ForkJoinPool-1-worker-2: I am on it!
VirtualThread[#20]/runnable@ForkJoinPool-1-worker-3: I am on it!
VirtualThread[#23]/runnable@ForkJoinPool-1-worker-1: I am on it!
VirtualThread[#24]/runnable@ForkJoinPool-1-worker-2: I am on it!
```

```
VirtualThread[#25]/runnable@ForkJoinPool-1-worker-1: I am on it!
----------------------------------------------------------------------
VirtualThread[#30,VT_1]/runnable@ForkJoinPool-1-worker-1: I am on it!
VirtualThread[#31,VT_2]/runnable@ForkJoinPool-1-worker-5: I am on it!
VirtualThread[#32,VT_3]/runnable@ForkJoinPool-1-worker-2: I am on it!
VirtualThread[#33,VT_4]/runnable@ForkJoinPool-1-worker-5: I am on it!
VirtualThread[#34,VT_5]/runnable@ForkJoinPool-1-worker-2: I am on it!
```

The `Executors` Utility Class

The `Executors` utility class provides the following methods to create executor services that implement the *one-thread-per-task* model of execution:

`java.util.concurrent.Executors`

```
static ExecutorService newVirtualThreadPerTaskExecutor()
static ExecutorService newThreadPerTaskExecutor(ThreadFactory tf)
```

The first method creates an `Executor` that starts a new *virtual* `Thread` for each task.

The second method creates an `Executor` that starts a new `Thread` for each task. The thread being a virtual or a platform thread depending on whether the thread factory is a virtual or a platform thread factory.

The first method is equivalent to the second method when the thread factory passed as parameter creates virtual threads.

The number of threads created by the `Executor` is unbounded.

Invoking `cancel(true)` on a `Future` representing the pending result of a task submitted to the `Executor` will interrupt the thread executing the task.

3.7 Scalability of Throughput with Virtual Threads

Figure 3.1b illustrates how a virtual thread executes its task by being mounted and unmounted on carrier threads during its lifetime. Example 3.7 demonstrates how virtual threads are executed by mounting on platform threads. In addition, it demonstrates the scalability of executing thousands of virtual threads.

The following line of code executed by a thread returns a string representation of the thread with pertinent information about the thread.

```
String vtInfo1 = Thread.currentThread().toString();
```

For a virtual thread, it returns pertinent information about the virtual thread in the following format:

```
VirtualThread[#22,VT_2]/runnable@ForkJoinPool-1-worker-2
```

From the string representation, we can see that the virtual thread with ID #22 is mounted on carrier thread worker-2. We can extract the name of carrier thread with this code:

```
String ctName1 = vtInfo1.substring(vtInfo1.indexOf('w'));  // worker-2
```

If virtual thread #22 executes a blocking operation, it will be unmounted and when it resumes execution, it might be mounted on a different or the same carrier thread. We can check this from the string representation of the virtual thread after resumption:

```
String vtInfo2 = Thread.currentThread().toString();
```

If the string representation of the virtual thread is as below, we know that it was mounted on carrier thread worker-4.

```
VirtualThread[#22,VT_2]/runnable@ForkJoinPool-1-worker-4
```

We can extract the name of carrier thread with this code:

```
String ctName2 = vtInfo2.substring(vtInfo2.indexOf('w'));  // worker-4
```

We can graphically represent the scheduling of a virtual thread from one carrier thread to another after a blocking operation as follows:

```
worker-2 -> worker-4
```

The getCarrierThreadName() static method at (7) in Example 3.7 extracts the name of the carrier thread the virtual thread is mounted on as outlined above. We call the getCarrierThreadName() method to extract the name of the carrier thread before and after each blocking operation in the task defined at (3):

```
String ctName1 = getCarrierThreadName();   // Carrier thread before.
someBlockingOperation();
String ctName2 = getCarrierThreadName();   // Carrier thread after.
...
```

The scheduling of a virtual thread from one carrier thread to another after each blocking operation is printed in the same order as the blocking operations. In order to keep the output manageable, only execution of a limited number of virtual threads is printed (controlled by the INTERVAL value defined at (2)).

A sleeping operation with a duration of 1 second is implemented as the blocking operation by the method someBlockingOperation() declared at (8).

The main() method at (9) uses a one-virtual-thread-per-task executor service to submit and execute the task a fixed number of times (NUM_OF_VT defined at (1)). The main() method also computes the time the executor service used to execute the submitted tasks (i.e., *the duration*) at (10) and *the throughput* (i.e., *number of tasks/duration*) at (11).

From the output in Example 3.7, we can see the carrier threads that a virtual thread was mounted on to execute the task. At the resumption of execution after blocking,

a virtual thread can be mounted on the same or a different carrier thread. Virtual thread #200000 was mounted consecutively on the same carrier thread twice:

```
Virtual Thread #200000: worker-2 -> worker-2 -> worker-5 -> worker-5
```

Whereas, virtual thread #300000 was mounted on different carrier threads after blocking during its lifetime.

```
Virtual Thread #300000: worker-3 -> worker-8 -> worker-4 -> worker-5
```

From the output in Example 3.7, we can see that the number of carrier threads employed by the executor service is 8 (the highest count on a carrier thread name in the output that is the same as the number of processors in this case).

Example 3.7 is running a million virtual threads (with a one-thread-per-task executor service taking a little over 54 seconds). The very high ratio of virtual threads executed to carrier threads employed to execute them results in formidable scaling of throughput, in this case a little over 18000 tasks/second. The curious reader is encouraged to experiment with different values for the number of tasks to execute, and to refactor the code to use platform threads and different executor services.

An important factor to keep in mind is that virtual threads can help to increase the *throughput* under the right circumstances, but they do not improve the *latency*—that is, they do not make each task execute faster.

Example 3.7 *Virtual Thread Execution and Scalability*

```
package vt;
import java.time.Duration;
import java.time.Instant;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.stream.IntStream;

public class VTExecutionDemo {

  public static final int NUM_OF_VT = 1_000_000;              // (1)
  public static final int INTERVAL = NUM_OF_VT/10;            // (2)

  // Create the task:                                            (3)
  static final Runnable task = () -> {

    // Obtain the names of carrier threads
    // the virtual thread was mounted on:                       (4)
    var ctName1 = getCarrierThreadName();
    someBlockingOperation();
    var ctName2 = getCarrierThreadName();
    someBlockingOperation();
    var ctName3 = getCarrierThreadName();
    someBlockingOperation();
    var ctName4 = getCarrierThreadName();

    // ID of this virtual thread:                               (5)
```

```
    var vtID = Thread.currentThread().threadId();

    // Print carrier threads the virtual thread was mounted on:   (6)
    if (vtID % INTERVAL == 0) {
      System.out.printf("Virtual Thread #%d: %s -> %s -> %s -> %s%n",
                        vtID, ctName1, ctName2, ctName3, ctName4);
    }
  };

  // Get the name of the carrier thread (format: worker-n)
  // the current virtual thread is mounted on.                    (7)
  static String getCarrierThreadName() {
    var vtInfo = Thread.currentThread().toString();
    return vtInfo.substring(vtInfo.indexOf('w'));
  }

  // A blocking operation to unmount a virtual thread.            (8)
  static void someBlockingOperation() {
    try {
      Thread.sleep(1000);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  public static void main(String[] args) {                     // (9)

    Instant start = Instant.now();

    try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
      IntStream.range(0, NUM_OF_VT).forEach(i -> executor.submit(task));
    }

    Instant finish = Instant.now();
    long duration = Duration.between(start, finish).toMillis();// (10)
    double throughput = (double) NUM_OF_VT / duration * 1000;  // (11)
    String format = """
        Number of virtual threads: %d
        Duration: %d ms
        Throughput: %.2f tasks/s
        """;
    System.out.printf(format, NUM_OF_VT, duration, throughput);
  }
}
```

Probable output from the program:

```
Virtual Thread #100000: worker-3 -> worker-7 -> worker-5 -> worker-7
Virtual Thread #200000: worker-2 -> worker-2 -> worker-5 -> worker-5
Virtual Thread #300000: worker-3 -> worker-8 -> worker-4 -> worker-5
Virtual Thread #400000: worker-2 -> worker-7 -> worker-8 -> worker-8
Virtual Thread #500000: worker-7 -> worker-4 -> worker-3 -> worker-2
Virtual Thread #600000: worker-1 -> worker-8 -> worker-4 -> worker-3
Virtual Thread #700000: worker-3 -> worker-7 -> worker-1 -> worker-8
Virtual Thread #800000: worker-2 -> worker-4 -> worker-5 -> worker-6
Virtual Thread #900000: worker-4 -> worker-4 -> worker-7 -> worker-8
```

```
Virtual Thread #1000000: worker-5 -> worker-5 -> worker-5 -> worker-7
Number of virtual threads: 1000000
Duration: 54407 ms
Throughput: 18379.99 tasks/s
```

3.8 Best Practices for Using Virtual Threads

In this section, we summarize some of the dos and don'ts of using virtual threads. Ultimately, *benchmarking* the performance of the concurrent application is the best way to determine any gains from using virtual threads. However, use of virtual threads boosts the throughput of one-thread-per-task-based applications under the following conditions:

- *Sufficiently large number of virtual threads*
- *Frequent short-lived blocking tasks*

These two conditions result in a high ratio of number of virtual threads to number of platform threads and the virtual threads being frequently unmounted so that their carrier threads can be scheduled to mount other virtual threads that are ready to execute their tasks, thereby effectively increasing the throughput of the application.

Avoid Pinning of Virtual Threads

As we have seen, a virtual thread is designed so that it can be unmounted from its carrier thread on executing a blocking operation, thereby allowing the JVM thread scheduler to mount another virtual thread on the carrier thread. However, there are situations where it is not possible to unmount a virtual thread from its associated carrier thread—called *pinning*. The virtual thread monopolizes its carrier thread, preventing it from servicing other virtual threads. Pinning of a virtual thread to its carrier thread can potentially impact both the scalability and the performance of a concurrent application, especially if more virtual threads become progressively pinned and thereby their associated carried threads cannot service other virtual threads.

Pinning of a virtual thread to its carrier thread can primarily occur in the following two contexts:

- *When the virtual thread is running code inside a synchronized block or method.*
- *When the virtual thread is calling a native method or a foreign function. (This topic is beyond the scope of this book and will not be discussed further.)*

Note that pinning does not render the application incorrect. Carrier threads have *bounded availability* (i.e., only a finite number of them can be created) and since pinning reduces the number of carrier threads available for executing virtual threads, pinning can have negative impact on the scalability of the application, especially if

it is frequent and long-lived. Note that a pinned virtual thread does *not block* its associated carrier thread unless a blocking operation is executed. If that happens, the associated carrier thread remains idle when blocked, further increasing the impact of pinning.

Example 3.8 illustrates both pinning of virtual threads in a synchronized block and how refactoring the code to use a reentrant lock can alleviate the problem. The example prints the schedule trace of carrier threads on which a virtual thread is mounted during the execution of its task.

In Example 3.8, a blocking operation is defined by the method `blockingOp()` at (3). The method returns a string of the form `"worker-n -> worker-m"` that identifies the scheduling of carrier threads the virtual thread was mounted on *before* and *after the blocking operation*, respectively.

Pinning in Synchronized Block

Example 3.8 defines a task at (4) that uses a *synchronized block* at (5) to implement a *critical region*. At the most only one thread can be executing the synchronized block. The code in the task traces the carrier threads that the virtual thread was mounted at various points in the code: before obtaining the lock of the synchronized object at (6), after obtaining the lock of the synchronized object at (7), executing the blocking operation at (8), and after the completion of the synchronized block at (9). Note that only one thread at a time can execute the synchronized block, other threads trying to obtain the lock of the synchronized object are blocked and have to wait their turn to execute the synchronized block.

If an application is run with the flag `jdk.tracePinnedThreads` having the value `short` or `full` on the command line, the JVM can print relevant stack trace information to identify a carrier thread that gets *blocked while its virtual thread is pinned*.

```
>java -Djdk.tracePinnedThreads=short MyApp
```

Example 3.8 is run with the above flag having the value `short` for tracing pinned threads. The scheduling trace of the carrier threads is printed at (10). For example, we see the following output for virtual thread `#28`:

```
Thread[#35,ForkJoinPool-1-worker-7,5,CarrierThreads]
    vt.VTPinningDemo.lambda$0(VTPinningDemo.java:41) <== monitors:1
[10:27:31] INFO: vt  #28: LockAcquiring(worker-7 -> worker-7) ->
                          BlockingOp(worker-7 -> worker-7) -> worker-7
```

The first two lines above show that carrier thread `#35`, having the name `worker-7`, was blocked during the execution of the blocking operation in the synchronized block. From the output we can see that virtual thread `#28` is pinned to carrier thread `#35` with the name `worker-7` that is blocked.

The last two lines show that virtual thread `#28` was pinned to carrier thread `worker-7` during the entire execution of the synchronized block: when acquiring the lock of the synchronized object, during the blocking operation, and after the synchronized block. It is important to note that not only was the virtual thread pinned to

its carrier thread, but the carrier thread was also blocked during the blocking operation. Pinning not only takes the associated carrier thread out of scheduling for other virtual threads, but during a blocking operation, it is also idle for the duration of the blocking period.

Similarly, pinning of the virtual threads can be traced in all runs of the task containing the synchronized block. Note that each virtual thread is assigned to a new carrier thread as virtual threads get pinned executing the synchronized block.

Example 3.8 *Pinning of Virtual Threads*

```
package vt;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;
import java.util.concurrent.locks.ReentrantLock;
import java.util.logging.Logger;
import java.util.stream.IntStream;

public class VTPinningDemo {
  private static final Logger logger =
      Logger.getLogger(VTPinningDemo.class.getName());
  static {
    System.setProperty("java.util.logging.SimpleFormatter.format",
        "[%1$tT] %4$s: %5$s%n");
  }

  public static final int NUMBER_OF_VT = 8;                              // (1)
  public static final int DURATION = 1000;                               // (2)

  // Blocking operation:
  private static String blockingOp() {                                   // (3)
    try {
      var ctNameBefore = getCarrierThreadName();
      TimeUnit.MILLISECONDS.sleep(DURATION);
      var ctNameAfter  = getCarrierThreadName();
      return String.format("%s -> %s", ctNameBefore, ctNameAfter);
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
    return "? -> ?";
  }

  // Task uses synchronized block:
  static final Runnable task1 = () -> {                                  // (4)
    String ctBeforeLock = "", ctAfterLock = "", ctAfterSynch = "",
           blockTrace = "";

    ctBeforeLock = getCarrierThreadName();                               // (5)
    synchronized (VTPinningDemo.class) {                                 // (6)
      ctAfterLock = getCarrierThreadName();                              // (7)
      blockTrace = blockingOp();                                         // (8)
    }
```

```
    ctAfterSynch  = getCarrierThreadName();                              // (9)

    logger.info(String.format(                                           // (10)
        "vt %4s: LockAcquiring(%s -> %s) -> BlockingOp(%s) -> %s",
        vtID(), ctBeforeLock, ctAfterLock, blockTrace, ctAfterSynch));
  };

  // Reentrant lock:
  public static final ReentrantLock lock = new ReentrantLock();          // (11)

  // Task uses reentrant lock:
  static final Runnable task2 = () -> {                                  // (12)
    String ctBeforeLock = "", ctAfterLock = "", ctAfterUnlock = "",
           blockTrace = "";

    ctBeforeLock = getCarrierThreadName();                               // (13)
    lock.lock();                                                         // (14)
    ctAfterLock = getCarrierThreadName();                                // (15)
    try {
      blockTrace = blockingOp();                                         // (16)
    } finally {
      lock.unlock();
    }
    ctAfterUnlock = getCarrierThreadName();                              // (17)

    logger.info(String.format(                                           // (18)
        "vt %4s: LockAcquiring(%s -> %s) -> BlockingOp(%s) -> %s",
        vtID(), ctBeforeLock, ctAfterLock, blockTrace, ctAfterUnlock
    ));
  };

  public static void main(String[] args) {                               // (19)
    logger.info("-----Synchronized block-----");
    try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
      IntStream.range(0, NUMBER_OF_VT).forEach(i -> executor.submit(task1));
    }

    logger.info("-----Reentrant lock-----");
    try (ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor()) {
      IntStream.range(0, NUMBER_OF_VT).forEach(i -> executor.submit(task2));
    }
  }

  static String vtID() {
    return  "#" + Thread.currentThread().threadId();
  }

  static String getCarrierThreadName() {
    var vtInfo = Thread.currentThread().toString();
    return vtInfo.substring(vtInfo.indexOf('w'));
  }
}
```

Probable output from the program (*edited to fit in page width*):

```
[10:27:30] INFO: -----Synchronized block-----
```

```
Thread[#35,ForkJoinPool-1-worker-7,5,CarrierThreads]
    vt.VTPinningDemo.lambda$0(VTPinningDemo.java:41) <== monitors:1
[10:27:31] INFO: vt  #28: LockAcquiring(worker-7 -> worker-7) ->
                          BlockingOp(worker-7 -> worker-7) -> worker-7
[10:27:32] INFO: vt  #23: LockAcquiring(worker-2 -> worker-2) ->
                          BlockingOp(worker-2 -> worker-2) -> worker-2
[10:27:33] INFO: vt  #29: LockAcquiring(worker-8 -> worker-8) ->
                          BlockingOp(worker-8 -> worker-8) -> worker-8
[10:27:35] INFO: vt  #26: LockAcquiring(worker-5 -> worker-5) ->
                          BlockingOp(worker-5 -> worker-5) -> worker-5
[10:27:36] INFO: vt  #21: LockAcquiring(worker-1 -> worker-1) ->
                          BlockingOp(worker-1 -> worker-1) -> worker-1
[10:27:37] INFO: vt  #27: LockAcquiring(worker-6 -> worker-6) ->
                          BlockingOp(worker-6 -> worker-6) -> worker-6
[10:27:38] INFO: vt  #24: LockAcquiring(worker-3 -> worker-3) ->
                          BlockingOp(worker-3 -> worker-3) -> worker-3
[10:27:39] INFO: vt  #25: LockAcquiring(worker-4 -> worker-4) ->
                          BlockingOp(worker-4 -> worker-4) -> worker-4
[10:27:39] INFO: -----Reentrant lock-----
[10:27:40] INFO: vt  #38: LockAcquiring(worker-4 -> worker-4) ->
                          BlockingOp(worker-4 -> worker-1) -> worker-1
[10:27:41] INFO: vt  #39: LockAcquiring(worker-3 -> worker-7) ->
                          BlockingOp(worker-7 -> worker-1) -> worker-1
[10:27:42] INFO: vt  #41: LockAcquiring(worker-1 -> worker-7) ->
                          BlockingOp(worker-7 -> worker-1) -> worker-1
[10:27:43] INFO: vt  #40: LockAcquiring(worker-6 -> worker-7) ->
                          BlockingOp(worker-7 -> worker-1) -> worker-1
[10:27:44] INFO: vt  #42: LockAcquiring(worker-5 -> worker-7) ->
                          BlockingOp(worker-7 -> worker-1) -> worker-1
[10:27:45] INFO: vt  #43: LockAcquiring(worker-8 -> worker-7) ->
                          BlockingOp(worker-7 -> worker-1) -> worker-1
[10:27:46] INFO: vt  #44: LockAcquiring(worker-2 -> worker-7) ->
                          BlockingOp(worker-7 -> worker-1) -> worker-1
[10:27:47] INFO: vt  #45: LockAcquiring(worker-7 -> worker-7) ->
                          BlockingOp(worker-7 -> worker-1) -> worker-1
```

Avoiding Pinning with a Reentrant Lock

Example 3.8 defines a task at (12) that uses a *reentrant lock* (declared at (11)) instead of a synchronized block to implement a critical region. The task uses the classical idiom for using a reentrant lock:

```
lock.lock();                    // Acquire the lock.
try {
  // Critical region
} finally {
  lock.unlock();                // Release the lock.
}
```

The method `lock()` of the `ReentrantLock` class is a *blocking* operation. Other threads will wait if the lock is already taken. Especially if a virtual thread gets blocked because the lock is taken, the virtual thread is unmounted and its carrier thread can

be scheduled to service other virtual threads by the JVM thread scheduler. There is no pinning involved.

As before, the code in the task traces the carrier threads that the virtual thread was mounted at various points in the code: before obtaining the lock at (13), after obtaining the lock at (15), executing the blocking operation at (16), and after the lock is freed at (17).

The scheduling trace of the carrier threads is printed at (18). For example, we see the following output for virtual thread `#38`:

```
[10:27:40] INFO: vt  #38: LockAcquiring(worker-4 -> worker-4) ->
                          BlockingOp(worker-4 -> worker-1) -> worker-1
```

The trace for acquiring the lock shows that virtual thread `#38` was mounted on carrier thread `worker-4`, most probably acquired the lock straight away, since the trace shows the same carrier thread before and after acquiring the lock.

The trace for the blocking operation shows that virtual thread `#38` was mounted on carrier thread `worker-4` before the blocking operation and was mounted on carrier thread `worker-1` when it was allowed to resume execution after blocking. While it was blocked, its carrier thread `worker-4` can be scheduled to execute other virtual threads. Again, there is no pinning during the blocking operation.

The `unlock()` method of the `ReentrantLock` class is *not* a blocking operation. The scheduling trace shows that virtual thread `#38` continued execution while mounted on carrier thread `worker-1`.

Similarly, the scheduling traces of the other virtual threads show that there is no pinning when using a reentrant lock in other runs of the task. Note that since the virtual threads were not pinned, their associated carrier threads can be scheduled to service other virtual threads waiting to execute, as evident from the runs where the same carrier thread was involved in the execution of several other virtual threads.

Avoid Using Virtual Threads for CPU-Bound Tasks

The benefit of virtual threads is best harnessed when virtual threads execute frequent short-lived blocking operations—that is the nature of virtual threads. In the Java APIs, I/O operations and blocking operations on relevant data structures have been refactored to unmount virtual threads, without any explicit action on the part of the application. Long-running CPU-intensive tasks will not unmount virtual threads, thus providing no additional advantage and are best executed by platform threads.

Avoid Pooling of Virtual Threads

A *thread pool* manages a *fixed* number of threads to limit the number of tasks that can execute concurrently—also called *limiting concurrency*. It does not create new threads, only allocating new tasks to existing threads as these become available.

Platform threads are a scarce resource, expensive to create and destroy. This is in contrast to virtual threads that are lightweight, cheap to create and destroy, specially designed for one-thread-per-task model of concurrent programming. Using a thread pool for virtual threads is thus inconsequential.

However, if it is necessary to limit the number of virtual threads that can execute concurrently, the interested reader should refer to the API of the `java.util.concurrent.Semaphore` class, as the Concurrency API does not provide any executor service that allows a fixed number of virtual threads.

Minimize Using Thread-Local Variables with Virtual Threads

A *thread-local variable* allows a thread to store a value that is only accessible in the scope of the thread where each thread has a private copy of the variable—thus ensuring its thread-safety.

Virtual threads work with thread-local variables, but as virtual threads can be created in the thousands, the sheer number of copies of each thread-local variable for each virtual thread can put a premium on memory space, especially if the data stored in the thread-local variables has a large memory footprint. This issue is less of a problem with platform threads, as these threads are seldom created in such large numbers as virtual threads.

For details on thread-local variables and their usage, the curious reader should refer to the API of the `java.lang.ThreadLocal<T>` class.

Avoid Substituting Virtual Threads for Platform Threads

Substituting virtual threads for platform threads is not always the answer to improve performance because virtual threads are *not* faster than platform threads. Under the right circumstances—large number of concurrent tasks that perform short-lived blocking operations—virtual threads can substantially increase the scalability of one-virtual-thread-per-task-based concurrent applications.

Future releases of Java aim to alleviate many of the issues regarding virtual threads that have been raised in this section.

Review Questions

3.1 Given the following code:

```
public class VTRQ1 {
```

```
  public static void main(String[] args) {
    Logger logger = Logger.getLogger("Test");
    Runnable r1 = () -> {
      int i = 0;
      while(true) {
        i++;
        logger.info(String.valueOf(i));
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          break;
        }
      }
    };
    Thread t1 = Thread.ofPlatform().name("acme").unstarted(r1);
    t1.start();
    t1.interrupt();
  }
}
```

Which scenario is possible when running the program?

(a) Program will log nothing and continue to run indefinitely.
(b) Program will log the value of `i` and continue to run indefinitely.
(c) Program will log nothing and terminate.
(d) Program will log one or more values of `i` and terminate.

3.2 Given the following code:

```
public class VTRQ2 {
    public static void main(String[] args) {
    Logger logger = Logger.getLogger("Test");
    Runnable r1 = () -> {
      int i = 0;
      while(true) {
        i++;
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          if (Thread.currentThread().isInterrupted()) {
            break;
          }
          logger.info(String.valueOf(i));
        }
      }
      logger.info(String.valueOf(i));
    };
    Thread t1 = Thread.ofPlatform().name("acme").unstarted(r1);
    t1.start();
    t1.interrupt();
  }
}
```

Which scenario is possible when running the program?

(a) Program will log nothing and continue to run indefinitely.

(b) Program will log the value of `i` and continue to run indefinitely.
(c) Program will log nothing and terminate.
(d) Program will log one or more values of `i` and terminate.

3.3 Given the following code:

```
public class VTRQ3 {
  public static void main(String[] args) {
    Logger logger = Logger.getLogger("Test");
    Runnable r1 = () -> {
      int i = 0;
      while(true) {
        i++;
        logger.info(String.valueOf(i));
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          break;
        }
      }
    };
    Thread t1 = Thread.ofVirtual().name("acme").unstarted(r1);
    t1.start();
    t1.interrupt();
  }
}
```

Which scenarios are possible when running the program?
Select the two correct answers.

(a) Program will log nothing and continue to run indefinitely.
(b) Program will log the value of `i` and continue to run indefinitely.
(c) Program will log nothing and terminate.
(d) Program will log one of more values of `i` and terminate.

3.4 Given the following code:

```
public class VTRQ4 {
  public static void main(String[] args) {
    Logger logger = Logger.getLogger("Test");
    Runnable r1 = () -> {
      int i = 0;
      while(true) {
        i++;
        try {
          Thread.sleep(1000);
        } catch (InterruptedException e) {
          if (Thread.currentThread().isInterrupted()) {
            break;
          }
          logger.info(String.valueOf(i));
        }
      }
      logger.info(String.valueOf(i));
```

```
        };
        Thread t1 = Thread.ofVirtual().name("acme").unstarted(r1);
        t1.start();
        t1.interrupt();
      }
    }
```

Which scenarios are possible when running the program?
Select the two correct answers.

(a) Program will log nothing and continue to run indefinitely.
(b) Program will log the value of `i` and continue to run indefinitely.
(c) Program will log nothing and terminate.
(d) Program will log one of more values of `i` and terminate.

3.5 Which statements are true about threads?
Select the two correct answers.

(a) The priority of a virtual thread cannot be changed.
(b) JVM only exits after all platform and virtual threads have completed their execution.
(c) When a virtual thread executes an I/O operation, it is blocked and its priority is set to 0.
(d) When a platform thread executes an I/O operation, it is blocked and its priority is set to 0.
(e) Virtual threads managed by a thread pool may improve application performance.
(f) Platform threads managed by a thread pool may improve application performance.

3.6 Which statements are true about virtual threads?
Select the three correct answers.

(a) Virtual threads are managed by the JVM rather than the operating system.
(b) Virtual threads can significantly improve the performance of CPU-bound tasks.
(c) Existing concurrency codebases using platform threads require minimal refactoring in order to leverage the benefits of virtual threads.
(d) Virtual threads can increase the throughput of a concurrency application as they drastically reduce the overhead associated with platform threads.
(e) Virtual threads are designed to replace platform threads in all Java concurrency applications.

3.7 Which statements are true about threads?
Select the two correct answers.

(a) A carrier thread is a virtual thread that is in the running state.
(b) A carrier thread is a platform thread on which a virtual thread is mounted for execution.
(c) A virtual thread has no name by default.

(d) An unmounted virtual thread when mounted to resume execution will continue running on the same platform thread.

3.8 Given the following code:

```
public class VTRQ8 {
  public static void main(String[] args) throws Exception {

    Runnable task = () -> System.out.printf("NAME: %s%n",
                                            Thread.currentThread().getName());
    // (1) Insert code here.
  }
}
```

Which options will cause the program to execute normally and always print `NAME: vt_1` when inserted at (1)?
Select the two correct answers.

(a)
```
Thread vt = Thread.startVirtualThread(task);
vt.setName("vt_1");
vt.start();
vt.join();
```

(b)
```
Thread vt = Thread.startVirtualThread(task);
vt.setName("vt_1");
vt.join();
```

(c)
```
Thread.Builder.OfVirtual vtb = Thread.ofVirtual().name("vt_", 1);
Thread vt = vtb.unstarted(task);
vt = vtb.start(task);
vt.join();
```

(d)
```
Thread.Builder.OfVirtual vtb = Thread.ofVirtual();
Thread vt = vtb.name("vt_", 1).started(task);
vt.join();
```

(e)
```
Thread.Builder.OfVirtual vtb = Thread.ofVirtual();
Thread vt = vtb.unstarted(task).name("vt_", 1);
vt.join();
```

(f)
```
Thread.Builder.OfVirtual vtb = Thread.ofVirtual().name("vt_", 1);
Thread vt = vtb.unstarted(task);
vt.start(task);
vt.join();
```

(g)
```
Thread.Builder.OfVirtual vtb = Thread.ofVirtual().name("vt_", 1);
Thread vt = vtb.unstarted(task);
vt = vt.start();
vt.join();
```

(h)
```
Thread vt = Thread.ofVirtual().name("vt_", 1).start(task);
vt.join();
```

(i)
```
Thread.ofVirtual().name("vt_", 1).start(task).join();
```

3.9 Given the following code:

```
public class VTRQ9 {
```

```
    public static void main(String[] args) throws Exception {

      Runnable task = () -> System.out.printf("NAME: %s%n",
                                              Thread.currentThread().getName());
      // (1) Insert code here.
    }
  }
```

Which options will cause the program to execute normally and print `NAME: vt_1` when inserted at (1)?
Select the two correct answers.

(a)
```
ThreadFactory vtf = Thread.ofVirtual().name("vt_0").factory();
Thread vt = vtf.newThread(task);
vt.setName("vt_1");
vt.start();
vt.join();
```

(b)
```
ThreadFactory vtf = Thread.ofVirtual().name("vt_0").factory();
Thread vt = new Thread(task);
vt.setName("vt_1");
vt.start();
vt.join();
```

(c)
```
Thread vt = Thread.ofVirtual().name("vt_1").factory().newThread(task);
vt.join();
```

(d)
```
Thread vt = Thread.ofVirtual().name("vt_1").factory().newThread(task);
vt.start().join();
```

(e)
```
Thread.Builder.OfVirtual vtb = Thread.ofVirtual().name("vt_", 1);
vtb.unstarted(task);
Thread vt = vtb.factory().newThread(task);
vt.start();
vt.join();
```

This page intentionally left blank

Part I: Index

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y