sufficiently small, they can be computed directly. The `compute()` method of `AverageGPATask` is responsible for both the division and the subsequent combination of these results to determine the overall average GPA.

Here's a simplified version of what the `AverageGPATask` class might look like:

```
class AverageGPATask extends RecursiveTask<Double> {
    private final List<Integer> studentGPAs;
    private final int start;
    private final int end;

    AverageGPATask(List<Integer> studentGPAs, int start, int end) {
        this.studentGPAs = studentGPAs;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Double compute() {
        // Split the task if it's too large, or compute directly if it's small enough
        // Combine the results of subtasks
    }
}
```

In this code, we create an instance of `AverageGPATask` to calculate the average GPA of all admitted students. We then use a `ForkJoinPool` to execute this task. The `invoke()` method of `ForkJoinPool` initiates the task and actively waits for its completion, akin to calling a `sort()` function and waiting for a sorted array, thereby returning the computed result. This is a great example of how the fork/join framework can be used to enhance the efficiency of a Java application.

Figure 1.5 shows a visual representation of the classes and their relationships. In this diagram, there are two classes: `FreshmenAdmissions` and `AverageGPATask`. The `FreshmenAdmissions` class has methods for setting admission information, getting approval information, and getting the approved name. The `AverageGPATask` class, which calculates the average GPA, has a `compute` method that returns a `Double`. The `ForkJoinPool` class uses the `AverageGPATask` class to calculate the average GPA; the `FreshmenAdmissions` class uses the `AverageGPATask` to calculate the average GPA of admitted students.

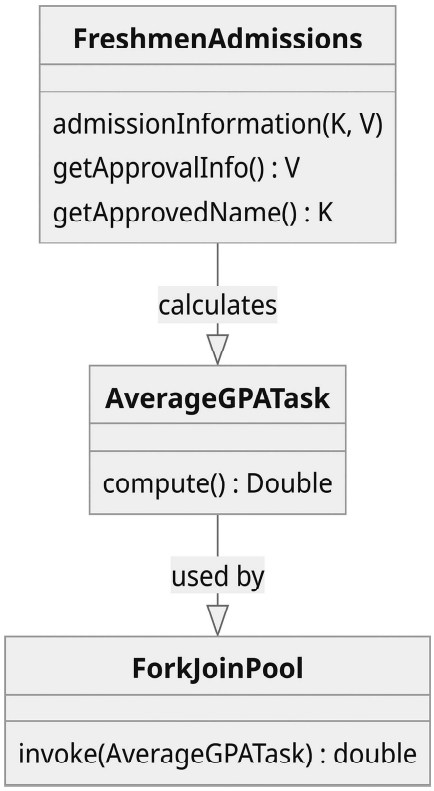

Figure 1.5 An Elaborate ForkJoinPool Example

Java SE 7 also brought significant enhancements to garbage collection. The parallel GC became NUMA-aware, optimizing performance on systems with the Non-Uniform Memory Access (NUMA) architecture. At its core, NUMA enhances memory access efficiency by taking into account the physical proximity of memory to a processing unit, such as a CPU or core. Prior to the introduction of NUMA, computers exhibited uniform memory access patterns, with all processing units sharing a single memory resource equally. NUMA introduced a paradigm in which processors access their local memory—memory that is physically closer to them—more swiftly than distant, non-local memory, which might be adjacent to other processors or shared among various cores. Understanding the intricacies of NUMA-aware garbage collection is vital for optimizing Java applications running on NUMA systems. (For an in-depth exploration of the NUMA-awareness in GC, see Chapter 6, "Advanced Memory Management and Garbage Collection in OpenJDK.")

In a typical NUMA setup, the memory access times can vary because data must traverse certain paths—or *hops*—within the system. These *hops* refer to the transfer of data between different nodes. Local memory access is direct and thus faster (fewer *hops*), while accessing non-local memory—residing farther from the processor—incurs additional *hops* resulting in increased latency due to the longer traversal paths. This architecture is particularly advantageous in

systems with multiple processors, as it maximizes the efficiency of memory utilization by leveraging the proximity factor.

NOTE My involvement with the implementation of NUMA-aware garbage collection first came during my tenure at AMD. I was involved in the performance characterization of AMD's Opteron processor,[16] a groundbreaking product that bought NUMA architecture into the mainstream. The Opteron[17] was designed with an integrated memory controller and a HyperTransport interconnect,[18] which allowed for direct, high-speed connections both between processors and between the processor and the memory. This design made the Opteron ideal for NUMA systems, and it was used in many high-performance servers, including Sun Microsystems' V40z.[19] The V40z was a high-performance server that was built by Sun Microsystems using AMD's Opteron processors. It was a NUMA system, so it could significantly benefit from NUMA-aware garbage collection. I was instrumental in providing the JDK team with data on cross-traffic and multi-JVM-local traffic, as well as insights into interleaved memory to illustrate how to amortize the latency of the *hops* between memory nodes.

To further substantiate the benefits of this feature, I implemented a prototype for the HotSpot VM with `-XX:+UseLargePages`. This prototype utilized large pages (also known as HugeTLB pages[20]) on Linux and the `numactl` API,[21] a tool for controlling NUMA policy on Linux systems. It served as a tangible demonstration of the potential performance improvements that could be achieved with NUMA-aware GC.

When the Parallel GC is NUMA-aware, it optimizes the way it handles memory to improve performance on NUMA systems. The young generation's eden space is allocated in NUMA-aware regions. In other words, when an object is created, it's placed in an eden space local to the processor that's executing the thread that created the object. This can significantly improve performance because accessing local memory is faster than accessing non-local memory.

The survivor spaces and the old generation are page interleaved in memory. Page interleaving is a technique used to distribute memory pages across different nodes in a NUMA system. It can help balance memory utilization across the nodes, leading to more efficient utilization of the memory bandwidth available on each node.

Note, however, that these optimizations are most beneficial on systems with a large number of processors and a significant disparity between local and non-local memory access times. On systems with a small number of processors or near-similar local and non-local memory access times, the impact of NUMA-aware garbage collection might be less noticeable. Also,

[16] Within AMD's Server Perf and Java Labs, my focus was on refining the HotSpot VM to enhance performance for Java workloads on NUMA-aware architectures, concentrating on JIT compiler optimizations, code generation, and GC efficiency for Opteron processor family.

[17] The Opteron was the first processor that supported AMD64, also known as x86-64ISA: https://en.wikipedia.org/wiki/Opteron.

[18] https://en.wikipedia.org/wiki/HyperTransport

[19] Oracle. "Introduction to the Sun Fire V20z and Sun Fire V40z Servers." https://docs.oracle.com/cd/E19121-01/sf.v40z/817-5248-21/chapter1.html.

[20] "HugeTLB Pages." www.kernel.org/doc/html/latest/admin-guide/mm/hugetlbpage.html.

[21] https://halobates.de/numaapi3.pdf

NUMA-aware garbage collection is not enabled by default; instead, it is enabled using the `-XX:+UseNUMA` JVM option.

In addition to the NUMA-aware GC, Java SE 7 Update 4 introduced the Garbage First Garbage Collector (G1 GC). G1 GC was designed to offer superior performance and predictability compared to its predecessors. Chapter 6 of this book delves into the enhancements brought by G1 GC, providing detailed examples and practical tips for its usage. For a comprehensive understanding of G1 GC, I recommend the *Java Performance Companion* book.[22]

Java 8 (Java SE 8)

Java 8 brought several significant features to the table that enhanced the Java programming language and its capabilities. These features can be leveraged to improve the efficiency and functionality of Java applications.

Language Features

One of the most notable features introduced in Java 8 was lambda expressions, which allow for more concise and functional-style programming. This feature can improve the simplicity of code by enabling more efficient implementation of functional programming concepts. For instance, if we have a list of students and we want to filter out those with a GPA greater than 80, we could use a lambda expression to do so concisely.

```
List<Student> admittedStudentsGPA = Arrays.asList(student1, student2, student3, student4,
student5, student6);
List<Student> filteredStudentsList = admittedStudentsGPA.stream().filter(s -> s.getGPA() > 80).
collect(Collectors.toList());
```

The lambda expression `s -> s.getGPA() > 80` is essentially a short function that takes a `Student` object `s` and returns a `boolean` value based on the condition `s.getGPA() > 80`. The `filter` method uses this lambda expression to filter the stream of students and `collect` gathers the results back into a list.

Java 8 also extended annotations to cover any place where a type is used, a capability known as *type annotations*. This helped enhance the language's expressiveness and flexibility. For example, we could use a type annotation to specify that the `studentGPAs` list should contain only non-null `Integer` objects:

```
private final List<@NonNull Integer> studentGPAs;
```

Additionally, Java 8 introduced the Stream API, which allows for efficient data manipulation and processing on collections. This addition contributed to JVM performance improvements, as developers could now optimize data processing more effectively. Here's an example using the Stream API to calculate the average GPA of the students in the `admittedStudentsGPA` list:

```
double averageGPA = admittedStudentsGPA.stream().mapToInt(Student::getGPA).average().orElse(0);
```

[22] www.pearson.com/en-us/subject-catalog/p/java-performance-companion/P200000009127/9780133796827

In this example, the `stream` method is called to create a stream from the list, `mapToInt` is used with a method reference `Student::getGPA` to convert each `Student` object to its GPA (an integer), `average` calculates the average of these integers, and `orElse` provides a default value of 0 if the stream is empty and an average can't be calculated.

Another enhancement in Java 8 was the introduction of default methods in interfaces, which enable developers to add new methods without breaking existing implementations, thus increasing the flexibility of interface design. If we were to define an interface for our tasks, we could use default methods to provide a standard implementation of certain methods. This could be useful if most tasks need to perform a set of common operations. For instance, we could define a `ComputableTask` interface with a `logStart` default method that logs when a task starts:

```
public interface Task {
    default void logStart() {
        System.out.println("Task started");
    }
    Double compute();
}
```

This `ComputableTask` interface could be implemented by any class that represents a task, providing a standard way to log the start of a task and ensuring that each task can be computed. For example, our `AverageGPATask` class could implement the `ComputableTask` interface:

```
class AverageGPATask extends RecursiveTask<Double> implements ComputableTask {
    // ...
}
```

JVM Enhancements

On the JVM front, Java 8 removed the Permanent Generation (PermGen) memory space, which was previously used to store metadata about loaded classes and other objects not associated with any instance. PermGen often caused problems due to memory leaks and the need for manual tuning. In Java 8, it was replaced by the Metaspace, which is located in native memory, rather than on the Java heap. This change eliminated many PermGen-related issues and improved the overall performance of the JVM. The Metaspace is discussed in more detail in Chapter 8, "Accelerating Time to Steady State with OpenJDK HotSpot VM."

Another valuable optimization introduced in JDK 8 update 20 was String Deduplication, a feature was specifically designed for the G1 GC. In Java, `String` objects are immutable, which means that once created, their content cannot be changed. This characteristic often leads to multiple `String` objects containing the same character sequences. While these duplicates don't impact the correctness of the application, they do contribute to increased memory footprint and can indirectly affect performance due to higher garbage collection overhead.

String Deduplication takes care of the issue of duplicate `String` objects by scanning the Java heap during GC cycles. It identifies duplicate strings and replaces them with references to a

single canonical instance. By decreasing the overall heap size and the number of live objects, the time taken for GC pauses can be reduced, contributing to lower tail latencies (this feature is discussed in more detail in Chapter 6). To enable the String Deduplication feature with the G1 GC, you can add the following JVM options:

```
-XX:+UseG1GC -XX:+UseStringDeduplication
```

Java 9 (Java SE 9) to Java 16 (Java SE 16)

Java 9: Project Jigsaw, JShell, AArch64 Port, and Improved Contended Locking

Java 9 brought several significant enhancements to the Java platform. The most notable addition was Project Jigsaw,[23] which implemented a module system, enhancing the platform's scalability and maintainability. We will take a deep dive into modularity in Chapter 3, "From Monolithic to Modular Java: A Retrospective and Ongoing Evolution."

Java 9 also introduced JShell, an interactive Java REPL (read–evaluate–print loop), enabling developers to quickly test and experiment with Java code. JShell allows for the evaluation of code snippets, including statements, expressions, and definitions. It also supports commands, which can be entered by adding a forward slash ("/") to the beginning of the command. For example, to change the interaction mode to verbose in JShell, you would enter the command `/set feedback verbose`.

Here's an example of how you might use JShell:

```
$ jshell
|  Welcome to JShell -- Version 17.0.7
|  For an introduction type: /help intro

jshell> /set feedback verbose
|  Feedback mode: verbose

jshell> boolean trueMorn = false
trueMorn ==> false
|  Created variable trueMorn : boolean

jshell> boolean Day(char c) {
   ...> return (c == 'Y');
   ...> }
|  Created method Day(char)

jshell> System.out.println("Did you wake up before 9 AM? (Y/N)")
Did you wake up before 9 AM? (Y/N)
```

[23] https://openjdk.org/projects/jigsaw/

```
jshell> trueMorn = Day((char) System.in.read())
Y
trueMorn ==> true
|  Assigned to trueMorn : boolean

jshell> System.out.println("It is " + trueMorn + " that you are a morning person")
It is true that you are a morning person
```

In this example, we first initialize `trueMorn` as `false` and then define our `Day()` method, which returns `true` or `false` based on the value of `c`. We then use `System.out.println` to ask our question and read the char input. We provide `Y` as input, so `trueMorn` is evaluated to be `true`.

JShell also provides commands like `/list`, `/vars`, `/methods`, and `/imports`, which provide useful information about the current JShell session. For instance, `/list` shows all the code snippets you've entered:

```
jshell> /list

   1 : boolean trueMorn = false;
   2 : boolean Day(char c) {
       return (c=='Y');
       }
   3 : System.out.println("Did you wake up before 9 AM? (Y/N)")
   4 : trueMorn = Day((char) System.in.read())
   5 : System.out.println("It is " + trueMorn + " that you are a morning person")
```

`/vars` lists all the variables you've declared:

```
jshell> /vars
|    boolean trueMorn = true
```

`/methods` lists all the methods you've defined:

```
jshell> /methods
|    boolean Day(char)
```

`/imports` shows all the imports in your current session:

```
jshell> /imports
|    import java.io.*
|    import java.math.*
|    import java.net.*
|    import java.nio.file.*
|    import java.util.*
|    import java.util.concurrent.*
```

```
|    import java.util.function.*
|    import java.util.prefs.*
|    import java.util.regex.*
|    import java.util.stream.*
```

The imports will also show up if you run the `/list –all` or `/list -start` command.

Furthermore, JShell can work with modules using the `--module-path` and `--add-modules` options, allowing you to explore and experiment with modular Java code.

Java 9 also added the AArch64 (also known as Arm64) port, thereby providing official support for the Arm 64-bit architecture in OpenJDK. This port expanded Java's reach in the ecosystem, allowing it to run on a wider variety of devices, including those with Arm processors.

Additionally, Java 9 improved contended locking through JEP 143: *Improve Contended Locking*.[24] This enhancement optimized the performance of intrinsic object locks and reduced the overhead associated with contended locking, improving the performance of Java applications that heavily rely on synchronization and contended locking. In Chapter 7, "Runtime Performance Optimizations: A Focus on Strings, Locks, and Beyond" we will take a closer look at locks and strings performance.

Release Cadence Change and Continuous Improvements

Starting from Java 9, the OpenJDK community decided to change the release cadence to a more predictable six-month cycle. This change was intended to provide more frequent updates and improvements to the language and its ecosystem. Consequently, developers could benefit from new features and enhancements more quickly.

Java 10: Local Variable Type Inference and G1 Parallel Full GC

Java 10 introduced local-variable type inference with the `var` keyword. This feature simplifies code by allowing the Java compiler to infer the variable's type from its initializer. This results in less verbose code, making it more readable and reducing boilerplate. Here's an example:

```
// Before Java 10
List<Student> admittedStudentsGPA = new ArrayList<>();

// With 'var' in Java 10
var admittedStudentsGPA = new ArrayList<Student>();
```

In this example, the `var` keyword is used to declare the variable `admittedStudentsGPA`. The compiler infers the type of `admittedStudentsGPA` from its initializer `new ArrayList <Student>()`, so you don't need to explicitly declare it.

[24] https://openjdk.org/jeps/143

```
// Using 'var' for local-variable type inference with admittedStudentsGPA list
var filteredStudentsList = admittedStudentsGPA.stream().filter(s -> s.getGPA() > 80).
collect(Collectors.toList());
```

In the preceding example, the `var` keyword is used in a more complex scenario involving a stream operation. The compiler infers the type of `filteredStudentsList` from the result of the stream operation.

In addition to addressing local-variable type inference, Java 10 improved the G1 GC by enabling parallel full garbage collections. This enhancement increased the efficiency of garbage collection, especially for applications with large heaps and those that suffer from some pathological (corner) case that causes evacuation failures, which in turn evoke the fallback full GC.

Java 11: New HTTP Client and String Methods, and Epsilon and Z GC

Java 11, a long-term support (LTS) release, introduced a variety of improvements to the Java platform, enhancing its performance and utility. One significant JVM addition in this release was the Epsilon GC,[25] an experimental no-op GC designed to test the performance of applications with minimal GC interference. The Epsilon GC allows developers to understand the impact of garbage collection on their application's performance, helping them make informed decisions about GC tuning and optimization.

```
$ java -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC -jar myApplication.jar
```

Epsilon GC is a unique addition to the JVM. It manages memory allocation, but doesn't implement any actual memory reclamation process. With this GC, once the available Java heap is used up, the JVM will shut down. Although this behavior might seem unusual, it's actually quite useful in certain scenarios—for instance, with extremely short-lived microservices. Moreover, for ultra-low-latency applications that can afford the heap footprint, Epsilon GC can help avoid all marking, moving, and compaction, thereby eliminating performance overhead. This streamlined behavior makes it an excellent tool for performance testing an application's memory pressures and understanding the overhead associated with VM/memory barriers.

Another noteworthy JVM enhancement in Java 11 was the introduction of the Z Garbage Collector (ZGC), which was initially available as an experimental feature. ZGC is a low-latency, scalable GC designed to handle large heaps with minimal pause times. By providing better garbage collection performance for applications with large memory requirements, ZGC enables developers to create high-performance applications that can effectively manage memory resources. (We'll take a deep dive into ZGC in Chapter 6 of this book.)

Keep in mind that these GCs are experimental in JDK 11, so using the `+UnlockExperimentalVMOptions` flag is necessary to enable them. Here's an example for ZGC:

```
$ java -XX:+UnlockExperimentalVMOptions -XX:+UseZGC -jar myApplication.jar
```

[25] https://blogs.oracle.com/javamagazine/post/epsilon-the-jdks-do-nothing-garbage-collector

In addition to JVM enhancements, Java 11 introduced a new HTTP client API that supports HTTP/2 and WebSocket. This API improved performance and provided a modern alternative to the `HttpURLConnection` API. As an example of its use, assume that we want to fetch some extracurricular activities' data related to our students from a remote API:

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder()
    .uri(URI.create("http://students-extracurricular.com/students"))
    .build();

client.sendAsync(request, HttpResponse.BodyHandlers.ofString())
    .thenApply(HttpResponse::body)
    .thenAccept(System.out::println);
```

In this example, we're making a GET request to `http://students-extracurricular.com/students` to fetch the extracurricular data. The response is handled asynchronously, and when the data is received, it's printed to the console. This new API provides a more modern and efficient way to make HTTP requests compared to the older `HttpURLConnection` API.

Furthermore, Java 11 added several new `String` methods, such as `strip()`, `repeat()`, and `isBlank()`. Let's look at our student GPA example and see how we can enhance it using these methods:

```
// Here we have student names with leading and trailing whitespaces
List<String> studentNames = Arrays.asList("  Monica  ", "  Ben  ", "  Annika  ", "  Bodin  ");

// We can use the strip() method to clean up the names
List<String> cleanedNames = studentNames.stream()
    .map(String::strip)
    .collect(Collectors.toList());

// Now suppose we want to create a String that repeats each name three times
// We can use the repeat() method for this
List<String> repeatedNames = cleanedNames.stream()
    .map(name -> name.repeat(3))
    .collect(Collectors.toList());

// And suppose we want to check if any name is blank after stripping out whitespaces
// We can use the isBlank() method for this
boolean hasBlankName = cleanedNames.stream()
    .anyMatch(String::isBlank);
```

These enhancements, along with numerous other improvements, made Java 11 a robust and efficient platform for developing high-performance applications.

Java 12: Switch Expressions and Shenandoah GC

Java 12 introduced the Shenandoah GC as an experimental feature. Like ZGC, Shenandoah GC is designed for large heaps with low latency requirements. This addition contributed to improved JVM performance for latency-sensitive applications. To enable Shenandoah GC, you can use the following JVM option:

```
$ java -XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC -jar myApplication.jar
```

Java 12 also introduced switch expressions as a preview feature, simplifying `switch` statements and making them more readable and concise:

```
String admissionResult = switch (status) {
    case "accepted" -> "Congratulations!";
    case "rejected" -> "Better luck next time.";
    default -> "Awaiting decision.";
};
```

In Java 14, you also have the option to use the `yield` keyword to return a value from a switch block. This is particularly useful when you have multiple statements in a case block. Here's an example:

```
String admissionResult = switch (status) {
    case "accepted" -> {
        // You can have multiple statements here
        yield "Congratulations!";
    }
    case "rejected" -> {
        // You can have multiple statements here
        yield "Better luck next time.";
    }
    default -> {
        // You can have multiple statements here
        yield "Awaiting decision.";
    }
};
```

Java 13: Text Blocks and ZGC Enhancements

Java 13 offered a number of new features and enhancements, including the introduction of text blocks as a preview feature. Text blocks simplify the creation of multiline string literals, making it easier to write and read formatted text:

```
String studentInfo = """
                     Name: Monica Beckwith
                     GPA: 3.83
                     Status: Accepted
                     """;
```

JDK 13 also included several enhancements to ZGC. For example, the new ZGC uncommit feature allowed unused heap memory to be returned to the operating system (OS) in a more timely and efficient manner. Prior to this feature, ZGC would release memory back to the OS only during a full GC cycle,[26] which could be infrequent and might result in large amounts of memory being held by the JVM unnecessarily. With the new feature, ZGC could return memory back to the operating system immediately upon detecting that a page is no longer needed by the JVM.

Java 14: Pattern Matching for instanceof and NUMA-Aware Memory Allocator for G1

Java 14 introduced pattern matching for the `instanceof` operator as a preview feature, simplifying type checking and casting:

```
if (object instanceof Student student) {
    System.out.println("Student name: " + student.getName());
}
```

Java 14 also introduced NUMA-aware memory allocation for the G1 GC. This feature is especially useful in multi-socket systems, where memory access latency and bandwidth efficiency can vary significantly across the sockets. With NUMA-aware allocation, G1 can allocate memory with a preference for the local NUMA node, which can significantly reduce memory access latency and improve performance. This feature can be enabled on supported platforms using the command-line option `-XX:+UseNUMA`.

Java 15: Sealed and Hidden Classes

Java 15 introduced two significant features that give developers more control over class hierarchies and the encapsulation of implementation details.

Sealed classes, introduced as a preview feature,[27] enable developers to define a limited set of subclasses for a superclass, providing more control over class hierarchies. For example, in a university application system, you could have:

```
public sealed class Student permits UndergraduateStudent, GraduateStudent, ExchangeStudent {
    private final String name;
    private final double gpa;
    // Common student properties and methods
}

final class UndergraduateStudent extends Student {
    // Undergraduate-specific properties and methods
}
```

[26] https://openjdk.org/jeps/351

[27] https://openjdk.org/jeps/360

```
final class GraduateStudent extends Student {
    // Graduate-specific properties and methods
}

final class ExchangeStudent extends Student {
    // Exchange-specific properties and methods
}
```

Here, `Student` is a sealed class, and only the specified classes (`UndergraduateStudent`, `GraduateStudent`, `ExchangeStudent`) can extend it, ensuring a controlled class hierarchy.

Hidden classes,[28] in contrast, are classes that are not discoverable by their name at runtime and can be used by frameworks that generate classes dynamically. For instance, a hidden class might be generated for a specific task like a query handler in a database framework:

```
// Within a database framework method
MethodHandles.Lookup lookup = MethodHandles.lookup();
Class<?> queryHandlerClass = lookup.defineHiddenClass(
    queryHandlerBytecode, true).lookupClass();
// The queryHandlerClass is now a hidden class, not directly usable by other classes.
```

The `defineHiddenClass` method generates a class from bytecode. Since it's not referenced elsewhere, the class remains internal and inaccessible to external classes.[29]

Both sealed and hidden classes enhance the Java language by providing tools for more precise control over class design, improving maintainability, and securing class implementation details.

Java 16: Records, ZGC Concurrent Thread-Stack Processing, and Port to Windows on Arm

Java 16 introduced records, a new class type that streamlines the creation of data-centric classes. Records provide a succinct syntax for defining immutable data structures:

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

Leading my team at Microsoft, we contributed to the Java ecosystem by enabling the JDK to function on Windows running on Arm 64 hardware. This significant undertaking was encapsulated in JEP 388: *Windows/AArch64 Port*,[30] where we successfully enabled the

[28] https://openjdk.org/jeps/371

[29] Although the hidden class example omits `MethodHandles.Lookup.ClassOption.NESTMATE` for simplicity, it's worth mentioning for a fuller understanding. `NESTMATE` enables a hidden class to access private members (like methods and fields) of its defining class, which is useful in advanced Java development. This specialized feature is not typically needed for everyday Java use but can be important for complex internal implementations.

[30] https://openjdk.org/jeps/388

template interpreter, the C1 and C2 JIT compilers, and all supported GCs. Our work broadened the environments in which Java can operate, further solidifying its position in the technology ecosystem.

In addition, ZGC in Java 15 evolved into a full supported feature. In Java 16, with JEP 376: *ZGC: Concurrent Thread-Stack Processing*,[31] ZGC underwent further enhancement. This improvement involves moving thread-stack processing to the concurrent phase, significantly reducing GC pause times—bolstering performance for applications managing large data sets.

Java 17 (Java SE 17)

Java 17, the most recent LTS release as of this book's writing, brings a variety of enhancements to the JVM, the Java language, and the Java libraries. These improvements significantly boost the efficiency, performance, and security of Java applications.

JVM Enhancements: A Leap Forward in Performance and Efficiency

Java 17 introduced several notable JVM improvements. JEP 356: *Enhanced Pseudo-Random Number Generators*[32] provides new interfaces and implementations for random number generation, offering a more flexible and efficient approach to generating random numbers in Java applications.

Another significant enhancement is JEP 382: *New macOS Rendering Pipeline*,[33] which improves the efficiency of Java two-dimensional graphics rendering on macOS. This new rendering pipeline leverages the native graphics capabilities of macOS, resulting in better performance and rendering quality for Java applications running on macOS.

Furthermore, JEP 391: *macOS/AArch64 Port*[34] adds a macOS/AArch64 port to the JDK. This port, supported by our Windows/AArch64 port, enables Java to run natively on macOS devices with M1-based architecture, further expanding the platform support for the JVM and enhancing performance on these architectures.

Security and Encapsulation Enhancements: Fortifying the JDK

Java 17 also sought to increase the security of the JDK. JEP 403: *Strongly Encapsulate JDK Internals*[35] makes it more challenging to access internal APIs that are not intended for general use. This change fortifies the security of the JDK and ensures compatibility with future releases by discouraging the use of internal APIs that may change or be removed in subsequent versions.

Another key security enhancement in Java 17 is the introduction of JEP 415: *Context-Specific Deserialization Filters*.[36] This feature provides a mechanism for defining deserialization filters based on context, offering a more granular control over the deserialization process. It addresses

[31] https://openjdk.org/jeps/376

[32] https://openjdk.org/jeps/356

[33] https://openjdk.org/jeps/382

[34] https://openjdk.org/jeps/391

[35] https://openjdk.org/jeps/403

[36] https://openjdk.org/jeps/415

some of the security concerns associated with Java's serialization mechanism, making it safer to use. As a result of this enhancement, application developers can now construct and apply filters to every deserialization operation, providing a more dynamic and context-specific approach compared to the static JVM-wide filter introduced in Java 9.

Language and Library Enhancements

Java 17 introduced several language and library enhancements geared toward improving developer productivity and the overall efficiency of Java applications. One key feature was JEP 406: *Pattern Matching for switch*,[37] which was introduced as a preview feature. This feature simplifies coding by allowing common patterns in `switch` statements and expressions to be expressed more concisely and safely. It enhances the readability of the code and reduces the chances of errors.

Suppose we're using a switch expression to generate a custom message for a student based on their status. The `String.format()` function is used to insert the student's name and GPA into the message:

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

// Let's assume we have a student
Student student = new Student("Monica", "accepted", 3.83);

String admissionResult = switch (student.status()) {
  case "accepted" -> String.format("%s has been accepted with a GPA of %.2f. Congratulations!",
student.name(), student.gpa());
  case "rejected" -> String.format("%s has been rejected despite a GPA of %.2f. Better luck next
time.", student.name(), student.gpa());
  case "waitlisted" -> String.format("%s is waitlisted. Current GPA is %.2f. Keep your fingers
crossed!", student.name(), student.gpa());
  case "pending" -> String.format("%s's application is still pending. Current GPA is %.2f. We hope
for the best!", student.name(), student.gpa());
  default -> String.format("The status of %s is unknown. Please check the application.", student.
name());
};

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

Moreover, Java 17 includes JEP 412: *Foreign Function and Memory API (Incubator)*,[38] which provides a pure Java API for calling native code and working with native memory. This API is designed to be safer, more efficient, and easier to use than the existing Java Native Interface (JNI). We will delve deeper into foreign function and memory API in Chapter 9, "Harnessing Exotic Hardware: The Future of JVM Performance Engineering."

[37] https://openjdk.org/jeps/406

[38] https://openjdk.org/jeps/412

Deprecations and Removals

Java 17 is also marked by the deprecation and removal of certain features that are no longer relevant or have been superseded by newer functionalities.

JEP 398: *Deprecate the Applet API for Removal*[39] deprecates the Applet API. This API has become obsolete, as most web browsers have removed support for Java browser plug-ins.

Another significant change is the removal of the experimental ahead-of-time (AOT) and JIT compiler associated with Graal, as per JEP 410: *Remove the Experimental AOT and JIT Compiler.*[40] The Graal compiler was introduced as an experimental feature in JDK 9 and was never intended to be a long-term feature of the JDK.

Lastly, JEP 411: *Deprecate the Security Manager for Removal*[41] made the Security Manager, a feature dating back to Java 1.0, no longer relevant and marked it for removal in a future release. The Security Manager was originally designed to protect the integrity of users' machines and the confidentiality of their data by running applets in a sandbox, which denied access to resources such as the file system or the network. However, with the decline of Java applets and the rise of modern security measures, the Security Manager has become less significant.

These changes reflect the ongoing evolution of the Java platform, as it continues to adapt to modern development practices and the changing needs of the developer community. Java 17 is unequivocally the best choice for developing robust, reliable, and high-performing applications. It takes the Java legacy to the next level by significantly improving performance, reinforcing security, and enhancing developer efficiency, thereby making it the undisputed leader in the field.

Embracing Evolution for Enhanced Performance

Throughout Java's evolution, numerous improvements to the language and the JVM have optimized performance and efficiency. Developers have been equipped with innovative features and tools, enabling them to write more efficient code. Simultaneously, the JVM has seen enhancements that boost its performance, such as new garbage collectors and support for different platforms and architectures.

As Java continues to evolve, it's crucial to stay informed about the latest language features, enhancements, and JVM improvements. By doing so, you can effectively leverage these advancements in your applications, writing idiomatic and pragmatic code that fully exploits the latest developments in the Java ecosystem.

In conclusion, the advancements in Java and the JVM have consistently contributed to the performance and efficiency of Java applications. For developers, staying up-to-date with these improvements is essential to fully utilize them in your work. By understanding and applying these new features and enhancements, you can optimize your Java applications' performance and ensure their compatibility with the latest Java releases.

---

[39] https://openjdk.org/jeps/398

[40] https://openjdk.org/jeps/410

[41] https://openjdk.org/jeps/411

Chapter 2

Performance Implications of Java's Type System Evolution

The type system is a fundamental aspect of any programming language, and Java is no exception. It governs how variables are declared, how they interact with each other, and how they can be used in your code. Java is known for its (strongly) static-typed nature. To better understand what that means, let's break down this concept:

- **Java is a statically type-checked language:** That means the variable types are checked (mostly) at compile time. In contrast, dynamically type-checked languages, like JavaScript, perform type checking at runtime. This static type-checking contributes to Java's performance by catching type errors at compile time, leading to more robust and efficient code.
- **Java is a strongly typed language:** This implies that Java will produce errors at compile time if the variable's declared type does not match the type of the value assigned to it. In comparison, languages like C may implicitly convert the value's type to match the variable type. For example, because C language performs implicit conversions, it can be considered a (weakly) static-typed language. In contrast, languages such as assembly languages are untyped.

Over the years, Java's type system has evolved significantly, with each new version bringing its own set of enhancements. This evolution has made the language not only more powerful and flexible, but also easier to use and understand. For instance, the introduction of generics in Java SE 5.0 allowed developers to write more type-safe code, reducing the likelihood of runtime type errors and improving the robustness of applications. The introduction of annotations in Java SE 5.0 provided a way for developers to embed metadata directly into their code; this metadata can be used by the JVM or other tools to optimize functionality or scalability. For example, the `@FunctionalInterface` annotation introduced in Java SE 8 indicates that an interface is intended for use with lambda expressions, aiding in code clarity.

The introduction of variable handle typed references in Java 9 provided a more efficient way to access variables, such as static or instance fields, array elements, and even off-heap areas, through their handles. These handles are typed references to their variables and maintain a minimum viable access set that is compatible with the current state of the Java Memory Model

and C/C++11 atomics. These standards define how threads interact through memory and which atomic operations they can perform. This feature enhances performance and scalability for applications requiring frequent variable access.

The introduction of switch expressions and sealed classes in Java 11 to Java 17 provided more expressive and concise ways to write code, reducing the likelihood of errors and improving the readability of the code. More recently, the ongoing work on Project Valhalla aims to introduce inline classes and generics over primitive types. These features are expected to significantly improve the scalability of Java applications by reducing the memory footprint of objects and enabling flattened data structures, which provide for better memory efficiency and performance by storing data in a more compact format, which both reduces memory usage and increases cache efficiency.

In Chapter 1, "The Performance Evolution of Java: The Language and the Virtual Machine," we explored the performance evolution of Java—the language and the JVM. With that context in mind, let's delve into Java's type system and build a similar timeline of its evolution.

Java's Primitive Types and Literals Prior to J2SE 5.0

Primitive data types, along with object types and arrays, form the foundation of the simplest type system in Java. The language has reserved keywords for these primitive types, such as `int`, `short`, `long`, and `byte` for integer types. Floating-point numbers use `float` and `double` keywords; `boolean` represents boolean types; and `char` is used for characters. String objects are essentially groups of characters and are immutable in nature; they have their own class in Java: `java.lang.String`. The various types of integers and floating-point numbers are differentiated based on their size and range, as shown in Table 2.1.

Table 2.1 **Java Primitive Data Types**

Data Types: Keywords	Size	Range*
char	16-bit Unicode	\u0000–\uffff
boolean	NA	true or false
byte	8-bit signed	$-128 < b \leq 127$
short	16-bit signed	$-32768 < s \leq 32767$
int	32-bit signed	(-2^{31})–$(2^{31} - 1)$*
long	64-bit	(-2^{63})–$(2^{63} - 1)$
float	32-bit single-precision	See [1]
double	64-bit double-precision	See [1]

* Range as defined prior to J2SE 5.0.

The use of primitive types in Java contributes to both performance and scalability of applications. Primitive types are more memory efficient than their wrapper class counterparts, which

are objects that encapsulate primitive types. Thus, the use of primitive types leads to more efficient code execution.

For some primitive types, Java can handle *literals*, which represent these primitives as specific values. Here are a few examples of acceptable literals prior to J2SE 5.0:

```
String wakie = "Good morning!";
char beforeNoon = 'Y';
boolean awake = true;
byte b = 127;
short s = 32767;
int i = 2147483647;
```

In the preceding examples, `"Good morning!"` is a string literal. Similarly, `'Y'` is a char literal, `true` is a boolean literal, and `127`, `32767`, and `2147483647` are `byte`, `short`, and `int` literals, respectively.

Another important literal in Java is *null*. While it isn't associated with any object or type, *null* is used to indicate an unavailable reference or an uninitialized state. For instance, the default value for a string object is `null`.

Java's Reference Types Prior to J2SE 5.0

Java reference types are quite distinct from the primitive data types. Before J2SE 5.0, there were three types of references: interfaces, instantiable classes, and arrays. Interfaces define functional specifications for attributes and methods, whereas instantiable classes (excluding utility classes, which mainly contain static methods and are not meant to be instantiated) implement interfaces to define attributes and methods and to describe object behaviors. Arrays are fixed-size data structures. The use of reference types in Java contributes to scalability of the Java applications, as using interfaces, instantiable classes, and arrays can lead to more modular and maintainable code.

Java Interface Types

Interfaces provide the foundation for methods that are implemented in classes that might otherwise be unrelated. Interfaces can specify code execution but cannot contain any executable code. Here's a simplified example:

```
public interface WakeupDB {
    int TIME_SYSTEM = 24;

    void setWakeupHr(String name, int hrDigits);

    int getWakeupHr(String name);

}
```

In this example, both `setWakeupHr` and `getWakeupHr` are abstract methods. As you can see, there are no implementation details in these abstract methods. We will revisit interface methods when we discuss Java 8. For now, since this section covers the evolution of types only up to J2SE 5.0, just know that each class implementing the interface must include the implementation details for these interface methods. For example, a class `DevWakeupDB` that implements `WakeupDB` will need to implement both the `setWakeupHr()` and `getWakeupHr()` methods, as shown here:

```
public class DevWakeupDB implements WakeupDB {
    //…

    public void SetWakeupHr(String name, int hrDigits) {
         //…
    }

    public int getWakeupHr(String name) {
         //…
    }

    //…
}
```

The earlier example also included a declaration for `TIME_SYSTEM`. A constant declared within an interface is implicitly considered `public`, `static`, and `final`, which means it is always accessible to all classes that implement the interface and is immutable.

Figure 2.1 is a simple class diagram in which `DevWakeupDB` is a class that implements the `WakeupDB` interface. The methods `setWakeupHr` and `getWakeupHr` are defined in the `WakeupDB` interface and implemented in the `DevWakeupDB` class. The `TIME_SYSTEM` attribute is also a part of the `WakeupDB` interface.

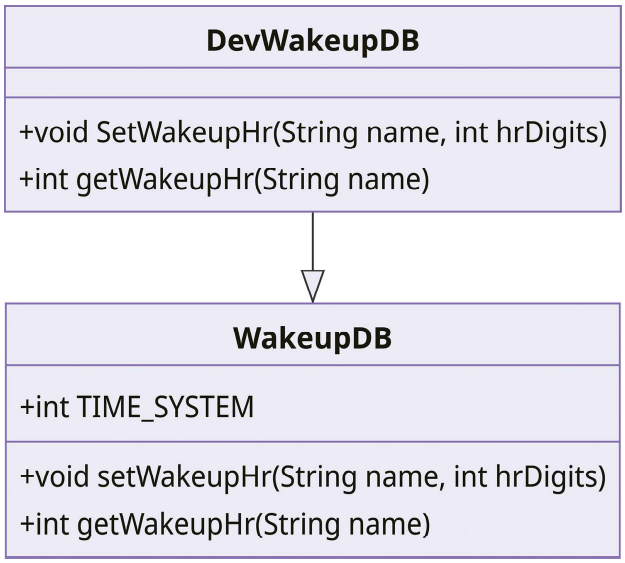

Figure 2.1 Class Diagram Showing Interface

Java Class Types

In the Java type system, a class defines a custom, aggregate data type that can contain heterogeneous data. A Java class encapsulates data and behavior and serves as a blueprint or template for creating objects. Objects have attributes (also known as fields or instance variables) that store information about the object itself. Objects also have methods, which are functions that can perform actions on or with the object's data. Let's look at an example using coding techniques prior to Java SE 5.0, with explicit field and object declarations:

```
class MorningPerson {

    private static boolean trueMorn = false;

    private static boolean Day(char c) {
        return (c == 'Y');
    }

    private static void setMornPers() throws IOException {
        System.out.println("Did you wake up before 9 AM? (Y/N)");
        trueMorn = Day((char) System.in.read());
    }

    private static void getMornPers() {
        System.out.println("It is " + trueMorn + " that you are a morning person");
    }

    public static void main(String[] args) throws IOException {
        setMornPers();
        getMornPers();
    }
}
```

This example shows a class `MorningPerson` with four methods: `Day()`, `setMornPers()`, `getMornPers()`, and `main()`. There is a static boolean class field called `trueMorn`, which stores the boolean value of a `MorningPerson` object. Each object will have access to this copy. The class field is initialized as follows:

```
private static boolean trueMorn = false;
```

If we didn't initialize the field, it would have been initialized to `false` by default, as all declared fields have their own defaults based on the data type.

Here are the two outputs depending on your input character:

```
$ java MorningPerson
Did you wake up before 9 AM? (Y/N)
N
```

```
It is false that you are a morning person

$ java MorningPerson
Did you wake up before 9 AM? (Y/N)
Y
It is true that you are a morning person
```

Java Array Types

Like classes, arrays in Java are considered aggregate data types. However, unlike classes, arrays store homogeneous data, meaning all elements in the array must be of the same type. Let's examine Java array types using straightforward syntax for their declaration and usage.

```
public class MorningPeople {

    static String[] peopleNames = {"Monica ", "Ben ", "Bodin ", "Annika"};
    static boolean[] peopleMorn = {false, true, true, false};

    private static void getMornPeople() {
        for (int i = 0; i < peopleMorn.length; i++) {
            if (peopleMorn[i]) {
                System.out.println(peopleNames[i] + "is a morning person");
            }
        }
    }

    public static void main (String[] args){
        getMornPeople();
    }

}
```

This example includes two single-dimensional arrays: a `String[]` array and a `boolean[]` array, both containing four elements. In the `for` loop, the `string.length` property is accessed, which provides the total count of elements in the array. The expected output is shown here:

```
Ben is a morning person
Bodin is a morning person
```

Arrays provide efficient access to multiple elements of the same type. So, for example, if we want to find early risers based on the day of the week, we could modify the single-dimensional boolean array by turning it into a multidimensional one. In this new array, one dimension (e.g., columns) would represent the days of the week, while the other dimension (rows) would represent the people. This structure can be visualized as shown in Table 2.2.

Table 2.2 **A Multidimensional Array**

	Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
Monica	False	False	True	True	True	True	False
Ben	False	True	True	True	True	True	True
Bodin	True	True	True	True	True	True	False
Annika	False	True	True	True	True	True	True

This would help us get more information, so, for example, we don't disturb Monica on Saturday, Sunday, or Monday mornings. 😊

Java's Type System Evolution from J2SE 5.0 until Java SE 8

J2SE 5.0 brought significant enhancements to the Java type system, including generics, autoboxing, annotations, and enumerations. Both annotations and enumerations were new reference types for Java.

Enumerations

Enumerations provide a way to represent a related set of objects. For instance, to list common environmental allergens in the Austin, Texas, area, we could define an enumeration as follows:

```
enum Season {
    WINTER, SPRING, SUMMER, FALL
}

enum Allergens {
    MOLD(Season.WINTER),
    CEDAR(Season.WINTER),
    OAK(Season.SPRING),
    RAGWEED(Season.FALL),
    DUST(Season.WINTER);

    private Season season;

    Allergens(Season season) {
        this.season = season;
    }

    public Season getSeason() {
        return season;
    }
}
```

Enumerations enhance the type system by providing a type-safe way to define a fixed set of related objects. This can improve code readability, maintainability, and safety. Additionally, the ability to add behavior and associated data to enumerations adds flexibility and expressiveness to your code.

Annotations

Annotations (also referred to as metadata) are modifiers that are preceded by an @ symbol. Users can create their own annotation types by declaring them much as they would declare an interface. Annotation types can have methods, known as annotation type elements. However, annotation type elements can't have any parameters. Let's look at an example:

```
public @interface BookProgress {
    int chapterNum();
    String startDate(); // Format: "YYYY-MM-DD"
    String endDate(); // Format: "YYYY-MM-DD"
    boolean submittedForReview() default false;
    int revNum() default 0;
    String[] reviewerNames(); // List of names of reviewers
    boolean[] reReviewNeeded(); // Corresponding to reviewerNames, indicates if re-review needed
}
```

In this example, we have defined an annotation type called `BookProgress`. Now we can use the `BookProgress` annotation type and provide values for the elements as shown here:

```
@BookProgress(
    chapterNum = 2,
    startDate = "2018-01-01", // Format: "YYYY-MM-DD"
    endDate = "2021-02-01",   // Format: "YYYY-MM-DD"
    submittedForReview = true,
    revNum = 1,
    reviewerNames = {"Ben", "Greg", "Charlie"},
    reReviewNeeded = {true, true, false} // Each boolean corresponds to a reviewer in the same order
)

public class Chapter2 {
        //chapter content …
}
```

Annotations enhance the type system by providing a way to attach metadata to various program elements. While the annotations themselves don't directly affect program performance, they enable tools and frameworks to perform additional checks, generate code, or provide runtime behavior. This can contribute to improved code quality, maintainability, and efficiency.

Other Noteworthy Enhancements (Java SE 8)

As discussed in Chapter 1, the Java language provides some predefined annotation types, such as `@SuppressWarnings`, `@Override`, and `@Deprecated`. Java SE 7 and Java SE 8 introduced two more predefined annotation types: `@SafeVarargs` and `@FunctionalInterface`, respectively. There are a few other annotations that apply to other annotations; they are called *meta-annotations*.

In Java SE 8, generics took a step further by helping the compiler infer the parameter type using "target-type" inference. Additionally, Java SE 8 introduced type annotations that extended the usage of annotations from just declarations to anywhere a type is used. Let's look at an example of how we could use Java SE 8's "generalized target-type inference" to our advantage:

```
HashMap<@NonNull String, @NonNull List<@Readonly String>> allergyKeys;
```

In this code snippet, the `@NonNull` annotation is used to indicate that the keys and values of the `allergyKeys` *HashMap* should never be null. This can help prevent `NullPointerExceptions` at runtime. The hypothetical `@Readonly` annotation on the `String` elements of the `List` indicates that these strings should not be modified. This can be useful in situations where you want to ensure the integrity of the data in your *HashMap*.

Building on these enhancements, Java SE 8 introduced another significant feature: lambda expressions. Lambda expressions brought a new level of expressiveness and conciseness to the Java language, particularly in the context of functional programming.

A key aspect of this evolution is the concept of "target typing." In the context of lambda expressions, target typing refers to the ability of the compiler to infer the type of a lambda expression from its target context. The target context can be a variable declaration, an assignment, a return statement, an array initializer, a method or constructor invocation, or a ternary conditional expression.

For example, we can sort the allergens based on their seasons using a lambda expression:

```
List<Allergens> sortedAllergens = Arrays.stream(Allergens.values())
    .sorted(Comparator.comparing(Allergens::getSeason))
    .collect(Collectors.toList());
```

In this example, the lambda expression `Allergens::getSeason` is a method reference that is equivalent to the lambda expression `allergen -> allergen.getSeason()`. The compiler uses the target type `Comparator<Allergens>` to infer the type of the lambda expression.

Java SE 7 and Java SE 8 introduced several other enhancements, such as the use of the underscore "_" character to enhance code readability by separating digits in numeric literals. Java SE 8 added the ability to use the `int` type for unsigned 32-bit integers (range: 0 to 232 – 1) and expanded the interface types to include static and default methods, in addition to method declarations. These enhancements benefit existing classes that implemented the interface by providing default and static methods with their own implementations.

Let's look at an example of how the interface enhancements can be used:

```
public interface MorningRoutine {
    default void wakeUp() {
        System.out.println("Waking up...");
    }

    void eatBreakfast();

    static void startDay() {
        System.out.println("Starting the day!");
    }
}

public class MyMorningRoutine implements MorningRoutine {
    public void eatBreakfast() {
        System.out.println("Eating breakfast...");
    }
}
```

In this example, `MyMorningRoutine` is a class that implements the `MorningRoutine` interface. It provides its own implementation of the `eatBreakfast()` method but inherits the default `wakeUp()` method from the interface. The `startDay()` method can be called directly from the interface, like so: `MorningRoutine.startDay();`.

The enhancements in Java SE 8 provided more flexibility and extensibility to interfaces. The introduction of static and default methods allowed interfaces to have concrete implementations, reducing the need for abstract classes in certain cases and supporting better code organization and reusability.

Java's Type System Evolution: Java 9 and Java 10

Variable Handle Typed Reference

The introduction of *VarHandles* in Java 9 provided developers with a new way to access variables through their handles, offering lock-free access without the need for `java.util.concurrent.atomic`. Previously, for fast memory operations like atomic updates or prefetching on fields and elements, core API developers or performance engineers often resorted to `sun.misc.Unsafe` for JVM *intrinsics*.

With the introduction of *VarHandles*, the *MethodHandles*[1] API was updated, and changes were made at the JVM and compiler levels to support *VarHandles*. *VarHandles* themselves are represented by the `java.lang.invoke.VarHandle` class, which utilizes a signature polymorphic method. Signature polymorphic methods can have multiple definitions based on the runtime types of the arguments. This enables dynamic dispatch, with the appropriate implementation

[1] "Class MethodHandles." https://docs.oracle.com/javase/9/docs/api/java/lang/invoke/MethodHandles.html.