Chapter 1

The Performance Evolution of Java: The Language and the Virtual Machine

More than three decades ago, the programming languages landscape was largely defined by C and its object-oriented extension, C++. In this period, the world of computing was undergoing a significant shift from large, cumbersome mainframes to smaller, more efficient minicomputers. C, with its suitability for Unix systems, and C++, with its innovative introduction of classes for object-oriented design, were at the forefront of this technological evolution.

However, as the industry started to shift toward more specialized and cost-effective systems, such as microcontrollers and microcomputers, a new set of challenges emerged. Applications were ballooning in terms of lines of code, and the need to "port" software to various platforms became an increasingly pressing concern. This often necessitated rewriting or heavily modifying the application for each specific target, a labor-intensive and error-prone process. Developers also faced the complexities of managing numerous static library dependencies and the demand for lightweight software on embedded systems—areas where C++ fell short.

It was against this backdrop that Java emerged in the mid-1990s. Its creators aimed to fill this niche by offering a "write once, run anywhere" solution. But Java was more than just a programming language. It introduced its own runtime environment, complete with a virtual machine (Java Virtual Machine [JVM]), class libraries, and a comprehensive set of tools. This all-encompassing ecosystem, known as the Java Development Kit (JDK), was designed to tackle the challenges of the era and set the stage for the future of programming. Today, more than a quarter of a century later, Java's influence in the world of programming languages remains strong, a testament to its adaptability and the robustness of its design.

The performance of applications emerged as a critical factor during this time, especially with the rise of large-scale, data-intensive applications. The evolution of Java's runtime system has played a pivotal role in addressing these performance challenges. Thanks to the optimization in generics, autoboxing and unboxing, and enhancements to the concurrency utilities, Java applications have seen significant improvements in both performance and scalability. Moreover, the changes have had far-reaching implications for the performance of the JVM itself. In

particular, the JVM has had to adapt and optimize its execution strategies to efficiently handle these new language features. As you read this book, bear in mind the historical context and the driving forces that led to Java's inception. The evolution of Java and its virtual machine have profoundly influenced the way developers write and optimize software for various platforms.

In this chapter, we will thoroughly examine the history of Java and JVM, highlighting the technological advancements and key milestones that have significantly shaped its development. From its early days as a solution for platform independence, through the introduction of new language features, to the ongoing improvements to the JVM, Java has evolved into a powerful and versatile tool in the arsenal of modern software development.

A New Ecosystem Is Born

In the 1990s, the internet was emerging, and web pages became more interactive with the introduction of Java applets. Java applets were small applications that ran within web browsers, providing a "real-time" experience for end users.

Applets were not only platform independent but also "secure," in the sense that the user needed to trust the applet writer. When discussing security in the context of the JVM, it's essential to understand that direct access to memory should be forbidden. As a result, Java introduced its own memory management system, called the garbage collector (GC).

> **NOTE** In this book, the acronym *GC* is used to refer to both *garbage collection,* the process of automatic memory management, and *garbage collector*, the module within the JVM that performs this process. The specific meaning will be clear based on the context in which *GC* is used.

Additionally, an abstraction layer, known as Java bytecode, was added to any executable. Java applets quickly gained popularity because their bytecode, residing on the web server, would be transferred and executed as its own process during web page rendering. Although the Java bytecode is platform independent, it is interpreted and compiled into native code specific to the underlying platform.

A Few Pages from History

The JDK included tools such as a Java compiler that translated Java code into Java bytecode. Java bytecode is the executable handled by the Java Runtime Environment (JRE). Thus, for different environments, only the runtime needed to be updated. As long as a JVM for a specific environment existed, the bytecode could be executed. The JVM and the GC served as the execution engines. For Java versions 1.0 and 1.1, the bytecode was interpreted to the native machine code, and there was no dynamic compilation.

Soon after the release of Java versions 1.0 and 1.1, it became apparent that Java needed to be more performant. Consequently, a just-in-time (JIT) compiler was introduced in Java 1.2. When

combined with the JVM, it provided dynamic compilation based on hot methods and loop-back branch counts. This new VM was called the Java HotSpot VM.

Understanding Java HotSpot VM and Its Compilation Strategies

The Java HotSpot VM plays a critical role in executing Java programs efficiently. It includes JIT compilation, tiered compilation, and adaptive optimization to improve the performance of Java applications.

The Evolution of the HotSpot Execution Engine

The HotSpot VM performs *mixed-mode* execution, which means that the VM starts in interpreted mode, with the bytecode being converted into native code based on a description table. The table has a template of native code corresponding to each bytecode instruction known as the *TemplateTable*; it is just a simple lookup table. The execution code is stored in a code cache (known as *CodeCache*). *CodeCache* stores native code and is also a useful cache for storing JIT-ted code.

> **NOTE** HotSpot VM also provides an interpreter that doesn't need a template, called the C++ interpreter. Some OpenJDK ports[1] choose this route to simplify porting of the VM to non-x86 platforms.

Performance-Critical Methods and Their Optimization

Performance engineering is a critical aspect of software development, and a key part of this process involves identifying and optimizing performance-critical methods. These methods are frequently executed or contain performance-sensitive code, and they stand to gain the most from JIT compilation. Optimizing performance-critical methods is not just about choosing appropriate data structures and algorithms; it also involves identifying and optimizing the methods based on their frequency of invocation, size and complexity, and available system resources.

Consider the following `BookProgress` class as an example:

```
import java.util.*;

public class BookProgress {
    private String title;
    private Map<String, Integer> chapterPages;
    private Map<String, Integer> chapterPagesWritten;

    public BookProgress(String title) {
        this.title = title;
```

[1] https://wiki.openjdk.org/pages/viewpage.action?pageId=13729802

```
        this.chapterPages = new HashMap<>();
        this.chapterPagesWritten = new HashMap<>();
    }

    public void addChapter(String chapter, int totalPages) {
        this.chapterPages.put(chapter, totalPages);
        this.chapterPagesWritten.put(chapter, 0);
    }

    public void updateProgress(String chapter, int pagesWritten) {
        this.chapterPagesWritten.put(chapter, pagesWritten);
    }

    public double getProgress(String chapter) {
        return ((double) chapterPagesWritten.get(chapter) / chapterPages.get(chapter)) * 100;
    }

    public double getTotalProgress() {
        int totalWritten = chapterPagesWritten.values().stream().mapToInt(Integer::intValue).sum();
        int total = chapterPages.values().stream().mapToInt(Integer::intValue).sum();
        return ((double) totalWritten / total) * 100;
    }
}

public class Main {
    public static void main(String[] args) {
        BookProgress book = new BookProgress("JVM Performance Engineering");
        String[] chapters = {
            "Performance Evolution",
            "Performance and Type System",
            "Monolithic to Modular",
            "Unified Logging System",
            "End-to-End Performance Optimization",
            "Advanced Memory Management",
            "Runtime Performance Optimization",
            "Accelerating Startup",
            "Harnessing Exotic Hardware"
        };
        for (String chapter : chapters) {
            book.addChapter(chapter, 100);
        }
        for (int i = 0; i < 50; i++) {
            for (String chapter : chapters) {
                int currentPagesWritten = book.chapterPagesWritten.get(chapter);
                if (currentPagesWritten < 100) {
                    book.updateProgress(chapter, currentPagesWritten + 2);
                    double progress = book.getProgress(chapter);
```

```
                    System.out.println("Progress for chapter " + chapter + ": " + progress + "%");
                }
            }
        }
        System.out.println("Total book progress: " + book.getTotalProgress() + "%");
    }
}
```

In this code, we've defined a `BookProgress` class to track the progress of writing a book, which is divided into chapters. Each chapter has a total number of pages and a current count of pages written. The class provides methods to add chapters, update progress, and calculate the progress of each chapter and the overall book.

The `Main` class creates a `BookProgress` object for a book titled "JVM Performance Engineering." It adds nine chapters, each with 100 pages, and simulates writing the book by updating the progress of each chapter in a round-robin fashion, writing two pages at a time. After each update, it calculates and prints the progress of the current chapter and, once all pages are written, the overall progress of the book.

The `getProgress(String chapter)` and `updateProgress(String chapter, int pagesWritten)` methods are identified as performance-critical methods. Their frequent invocation makes them prime candidates for optimization by the HotSpot VM, illustrating how certain methods in a program may require more attention for performance optimization due to their high frequency of use.

Interpreter and JIT Compilation

The HotSpot VM provides an interpreter that converts bytecode into native code based on the *TemplateTable*. Interpretation is the first step in adaptive optimization offered by this VM and is considered the slowest form of bytecode execution. To make the execution faster, the HotSpot VM utilizes adaptive JIT compilation. The JIT-optimized code replaces the template code for methods that are identified as performance critical.

As mentioned in the previous section, the HotSpot VM monitors executed code for performance-critical methods based on two key metrics—method entry counts and loop-back branch counts. The VM assigns call counters to individual methods in the Java application. When the entry count exceeds a preestablished value, the method or its callee is chosen for asynchronous JIT compilation. Similarly, there is a counter for each loop in the code. Once the HotSpot VM determines that the loop-back branches (also known as loop-back edges) have crossed their threshold, the JIT optimizes that particular loop. This optimization is called on-stack replacement (OSR). With OSR, only the loop for which the loop-back branch counter overflowed will be compiled and replaced asynchronously on the execution stack.

Print Compilation

A very handy command-line option that can help us better understand adaptive optimization in the HotSpot VM is `-XX:+PrintCompilation`. This option also returns information on different optimized compilation levels, which are provided by an adaptive optimization called tiered compilation (discussed in the next subsection).

The output of the `-XX:+PrintCompilation` option is a log of the HotSpot VM's compilation tasks. Each line of the log represents a single compilation task and includes several pieces of information:

- The timestamp in milliseconds since the JVM started and this compilation task was logged.
- The unique identifier for this compilation task.
- Flags indicating certain properties of the method being compiled, such as whether it's an OSR method (%), whether it's synchronized (s), whether it has an exception handler (!), whether it's blocking (b), or whether it's native (n).
- The tiered compilation level, indicating the level of optimization applied to this method.
- The fully qualified name of the method being compiled.
- For OSR methods, the bytecode index where the compilation started. This is usually the start of a loop.
- The size of the method in the bytecode, in bytes.

Here are a few examples of the output of the `-XX:+PrintCompilation` option:

```
567  693 % !   3    org.h2.command.dml.Insert::insertRows @ 76 (513 bytes)
656  797     n 0    java.lang.Object::clone (native)
779  835  s    4    java.lang.StringBuffer::append (13 bytes)
```

These logs provide valuable insights into the behavior of the HotSpot VM's adaptive optimization, helping us understand how our Java applications are optimized at runtime.

Tiered Compilation

Tiered compilation, which was introduced in Java 7, provides multiple levels of optimized compilations, ranging from T0 to T4:

1. **T0:** Interpreted code, devoid of compilation. This is where the code starts and then moves on to the T1, T2, or T3 level.
2. **T1–T3:** Client-compiled mode. T1 is the first step where the method invocation counters and loop-back branch counters are used. At T2, the client compiler includes profiling information, referred to as profile-guided optimization; it may be familiar to readers who are conversant in static compiler optimizations. At the T3 compilation level, completely profiled code can be generated.
3. **T4:** The highest level of optimization provided by the HotSpot VM's server compiler.

Prior to tiered compilation, the server compiler would employ the interpreter to collect such profiling information. With the introduction of tiered compilation, the code reaches client compilation levels faster, and now the profiling information is generated by client-compiled methods themselves, providing better start-up times.

NOTE Tiered compilation has been enabled by default since Java 8.

Client and Server Compilers

The HotSpot VM provides two flavors of compilers: the fast client compiler (also known as the C1 compiler) and the server compiler (also known as the C2 compiler).

1. **Client compiler (C1):** Aims for fast start-up times in a client setup. The JIT invocation thresholds are lower for a client compiler than for a server compiler. This compiler is designed to compile code quickly, providing a fast start-up time, but the code it generates is less optimized.
2. **Server compiler (C2):** Offers many more adaptive optimizations and better thresholds geared toward higher performance. The counters that determine when a method/loop needs to be compiled are still the same, but the invocation thresholds are different (much lower) for a client compiler than for a server compiler. The server compiler takes longer to compile methods but produces highly optimized code that is beneficial for long-running applications. Some of the optimizations performed by the C2 compiler include *inlining* (replacing method invocations with the method's body), loop unrolling (increasing the loop body size to decrease the overhead of loop checks and to potentially apply other optimizations such as loop vectorization), dead code elimination (removing code that does not affect the program results), and range-check elimination (removing checks for index out-of-bounds errors if it can be assured that the array index never crosses its bounds). These optimizations help to improve the execution speed of the code and reduce the overhead of certain operations.[2]

Segmented Code Cache

As we delve deeper into the intricacies of the HotSpot VM, it's important to revisit the concept of the code cache. Recall that the code cache is a storage area for native code generated by the JIT compiler or the interpreter. With the introduction of tiered compilation, the code cache also becomes a repository for profiling information gathered at different levels of tiered compilation. Interestingly, even the *TemplateTable*, which the interpreter uses to look up the native code sequence for each bytecode, is stored in the code cache.

The size of the code cache is fixed at start-up but can be modified on the command line by passing the desired maximum value to `-XX:ReservedCodeCacheSize`. Prior to Java 7, the default value for this size was 48 MB. Once the code cache was filled up, all compilation would cease. This posed a significant problem when tiered compilation was enabled, as the code cache would contain not only JIT-compiled code (represented as *nmethod* in the HotSpot VM) but also profiled code. The *nmethod* refers to the internal representation of a Java method that has been compiled into machine code by the JIT compiler. In contrast, the profiled code is the code that has been analyzed and optimized based on its runtime behavior. The code cache needs

[2] "What the JIT!? Anatomy of the OpenJDK HotSpot VM." infoq.com.

to manage both of these types of code, leading to increased complexity and potential performance issues.

To address these problems, the default value for `ReservedCodeCacheSize` was increased to 240 MB in JDK 7 update 40. Furthermore, when the code cache occupancy crosses a preset `CodeCacheMinimumFreeSpace` threshold, the JIT compilation halts and the JVM runs a *sweeper*. The *nmethod* sweeper reclaims space by evacuating older compilations. However, sweeping the entire code cache data structure can be time-consuming, especially when the code cache is large and nearly full.

Java 9 introduced a significant change to the code cache: It was segmented into different regions based on the type of code. This not only reduced the sweeping time but also minimized fragmentation of the long-lived code by shorter-lived code. Co-locating code of the same type also reduced hardware-level instruction cache misses.

The current implementation of the segmented code cache includes the following regions:

- **Non-method code heap region:** This region is reserved for VM internal data structures that are not related to Java methods. For example, the *TemplateTable*, which is a VM internal data structure, resides here. This region doesn't contain compiled Java methods.
- **Non-profiled *nmethod* code heap:** This region contains Java methods that have been compiled by the JIT compiler without profiling information. These methods are fully optimized and are expected to be long-lived, meaning they won't be recompiled frequently and may need to be reclaimed only infrequently by the sweeper.
- **Profiled *nmethod* code heap:** This region contains Java methods that have been compiled with profiling information. These methods are not as optimized as those in the non-profiled region. They are considered transient because they can be recompiled into more optimized versions and moved to the non-profiled region as more profiling information becomes available. They can also be reclaimed by the sweeper as often as needed.

Each of these regions has a fixed size that can be set by their respective command-line options:

Heap Region Type	Size Command-Line Option
Non-method code heap	`-XX:NonMethodCodeHeapSize`
Non-profiled *nmethod* code heap	`-XX:NonProfiledCodeHeapSize`
Profiled *nmethod* code heap	`-XX:ProfiledCodeHeapSize`

Going forward, the hope is that the segmented code caches can accommodate additional code regions for heterogeneous code such as ahead-of-time (AOT)–compiled code and code for hardware accelerators.[3] There's also the expectation that the fixed sizing thresholds can be upgraded to utilize adaptive resizing, thereby avoiding wastage of memory.

[3] JEP 197: *Segmented Code Cache.* https://openjdk.org/jeps/197.

Adaptive Optimization and Deoptimization

Adaptive optimization allows the HotSpot VM runtime to optimize the interpreted code into compiled code or insert an optimized loop on the stack (so we could have something like an "interpreted to compiled, and back to interpreted" code execution sequence). There is another major advantage of adaptive optimization, however—in deoptimization of code. That means the compiled code could go back to being interpreted, or a higher-optimized code sequence could be rolled back into a less-optimized sequence.

Dynamic deoptimization helps Java reclaim code that may no longer be relevant. A few example use cases are when checking interdependencies during dynamic class loading, when dealing with polymorphic call sites, and when reclaiming less-optimized code. Deoptimization will first make the code "not entrant" and eventually reclaim it after marking it as "zombie" code.[4]

Deoptimization Scenarios

Deoptimization can occur in several scenarios when working with Java applications. In this section, we'll explore two of these scenarios.

Class Loading and Unloading

Consider an application containing two classes, `Car` and `DriverLicense`. The `Car` class requires a `DriverLicense` to enable drive mode. The JIT compiler optimizes the interaction between these two classes. However, if a new version of the `DriverLicense` class is loaded due to changes in driving regulations, the previously compiled code may no longer be valid. This necessitates deoptimization to revert to the interpreted mode or a less-optimized state. This allows the application to employ the new version of the `DriverLicense` class.

Here's an example code snippet:

```
class Car {
    private DriverLicense driverLicense;

    public Car(DriverLicense driverLicense) {
        this.driverLicense = driverLicense;
    }

    public void enableDriveMode() {
        if (driverLicense.isAdult()) {
            System.out.println("Drive mode enabled!");
        } else if (driverLicense.isTeenDriver()) {
            if (driverLicense.isLearner()) {
                System.out.println("You cannot drive without a licensed adult's supervision.");
            } else {
                System.out.println("Drive mode enabled!");
            }
        } else {
```

[4] https://www.infoq.com/articles/OpenJDK-HotSpot-What-the-JIT/

```
            System.out.println("You don't have a valid driver's license.");
        }
    }
}

class DriverLicense {
    private boolean isTeenDriver;
    private boolean isAdult;
    private boolean isLearner;

    public DriverLicense(boolean isTeenDriver, boolean isAdult, boolean isLearner) {
        this.isTeenDriver = isTeenDriver;
        this.isAdult = isAdult;
        this.isLearner = isLearner;
    }

    public boolean isTeenDriver() {
        return isTeenDriver;
    }

    public boolean isAdult() {
        return isAdult;
    }

    public boolean isLearner() {
        return isLearner;
    }
}

public class Main {
    public static void main(String[] args) {
        DriverLicense driverLicense = new DriverLicense(false, true, false);
        Car myCar = new Car(driverLicense);
        myCar.enableDriveMode();
    }
}
```

In this example, the `Car` class requires a `DriverLicense` to enable drive mode. The driver's license can be for an adult, a teen driver with a learner's permit, or a teen driver with a full license. The `enableDriveMode()` method checks the driver's license using the `isAdult()`, `isTeenDriver()`, and `isLearner()` methods, and prints the appropriate message to the console.

If a new version of the `DriverLicense` class is loaded, the previously optimized code may no longer be valid, triggering deoptimization. This allows the application to use the new version of the `DriverLicense` class without any issues.

Polymorphic Call Sites

Deoptimization can also occur when working with polymorphic call sites, where the actual method to be invoked is determined at runtime. Let's look at an example using the `DriverLicense` class:

```
abstract class DriverLicense {
    public abstract void drive();
}

class AdultLicense extends DriverLicense {
    public void drive() {
        System.out.println("Thanks for driving responsibly as an adult");
    }
}

class TeenPermit extends DriverLicense {
    public void drive() {
        System.out.println("Thanks for learning to drive responsibly as a teen");
    }
}

class SeniorLicense extends DriverLicense {
    public void drive() {
        System.out.println("Thanks for being a valued senior citizen");
    }
}

public class Main {
    public static void main(String[] args) {
        DriverLicense license = new AdultLicense();
        license.drive(); // monomorphic call site

        // Changing the call site to bimorphic
        if (Math.random() < 0.5) {
            license = new AdultLicense();
        } else {
            license = new TeenPermit();
        }
        license.drive(); // bimorphic call site

        // Changing the call site to megamorphic
        for (int i = 0; i < 100; i++) {
            if (Math.random() < 0.33) {
                license = new AdultLicense();
```

```
            } else if (Math.random() < 0.66) {
                license = new TeenPermit();
            } else {
                license = new SeniorLicense();
            }
            license.drive(); // megamorphic call site
        }
    }
}
```

In this example, the abstract `DriverLicense` class has three subclasses: `AdultLicense`, `TeenPermit`, and `SeniorLicense`. The `drive()` method is overridden in each subclass with different implementations.

First, when we assign an `AdultLicense` object to a `DriverLicense` variable and call `drive()`, the HotSpot VM optimizes the call site to a monomorphic call site and caches the target method address in an *inline cache* (a structure to track the call site's type profile).

Next, we change the call site to a bimorphic call site by randomly assigning an `AdultLicense` or `TeenPermit` object to the `DriverLicense` variable and calling `drive()`. Because there are two possible types, the VM can no longer use the monomorphic dispatch mechanism, so it switches to the bimorphic dispatch mechanism. This change does not require deoptimization—and still provides a performance boost by reducing the number of virtual method dispatches needed at the call site.

Finally, we change the call site to a megamorphic call site by randomly assigning an `AdultLicense`, `TeenPermit`, or `SeniorLicense` object to the `DriverLicense` variable and calling `drive()` 100 times. As there are now three possible types, the VM cannot use the bimorphic dispatch mechanism and must switch to the megamorphic dispatch mechanism. This change also does not require deoptimization.

However, if we were to introduce a new subclass `InternationalLicense` and change the call site to include it, the VM could potentially deoptimize the call site and switch to a megamorphic or polymorphic call site to handle the new type. This change is necessary because the VM's type profiling information for the call site would be outdated, and the previously optimized code would no longer be valid.

Here's the code snippet for the new subclass and the updated call site:

```
class InternationalLicense extends DriverLicense {
    public void drive() {
        System.out.println("Thanks for driving responsibly as an international driver");
    }
}

// Updated call site
for (int i = 0; i < 100; i++) {
```

```
    if (Math.random() < 0.25) {
        license = new AdultLicense();
    } else if (Math.random() < 0.5) {
        license = new TeenPermit();
    } else if (Math.random() < 0.75) {
        license = new SeniorLicense();
    } else {
        license = new InternationalLicense();
    }
    license.drive(); // megamorphic call site with a new type
}
```

HotSpot Garbage Collector: Memory Management Unit

A crucial component of the HotSpot execution engine is its memory management unit, commonly known as the garbage collector (GC). HotSpot provides multiple garbage collection algorithms that cater to a trifecta of performance aspects: application responsiveness, throughput, and overall footprint. *Responsiveness* refers to the time taken to receive a response from the system after sending a stimulus. *Throughput* measures the number of operations that can be performed per second on a given system. *Footprint* can be defined in two ways: as optimizing the amount of data or objects that can fit into the available space and as removing redundant information to save space.

Generational Garbage Collection, Stop-the-World, and Concurrent Algorithms

OpenJDK offers a variety of generational GCs that utilize different strategies to manage memory, with the common goal of improving application performance. These collectors are designed based on the principle that "most objects die young," meaning that most newly allocated objects on the Java heap are short-lived. By taking advantage of this observation, generational GCs aim to optimize memory management and significantly reduce the negative impact of garbage collection on the performance of the application.

Heap collection in GC terms involves identifying live objects, reclaiming space occupied by garbage objects, and, in some cases, compacting the heap to reduce fragmentation. Fragmentation can occur in two ways: (1) internal fragmentation, where allocated memory blocks are larger than necessary, leaving wasted space within the blocks; and (2) external fragmentation, where memory is allocated and deallocated in such a way that free memory is divided into noncontiguous blocks. External fragmentation can lead to inefficient memory use and potential allocation failures. Compaction is a technique used by some GCs to combat external fragmentation; it involves moving objects in memory to consolidate free memory into a single contiguous block. However, compaction can be a costly operation in terms of CPU usage and can cause lengthy pause times if it's done as a stop-the-world operation.

The OpenJDK GCs employ several different GC algorithms:

- **Stop-the-world (STW) algorithms:** STW algorithms pause application threads for the entire duration of the garbage collection work. Serial, Parallel, (Mostly) Concurrent Mark and Sweep (CMS), and Garbage First (G1) GCs use STW algorithms in specific phases of their collection cycles. The STW approach can result in longer pause times when the heap fills up and runs out of allocation space, especially in nongenerational heaps, which treat the heap as a single continuous space without segregating it into generations.
- **Concurrent algorithms:** These algorithms aim to minimize pause times by performing most of their work concurrently with the application threads. CMS is an example of a collector using concurrent algorithms. However, because CMS does not perform compaction, fragmentation can become an issue over time. This can lead to longer pause times or even cause a fallback to a full GC using the Serial Old collector, which does include compaction.
- **Incremental compacting algorithms:** The G1 GC introduced incremental compaction to deal with the fragmentation issue found in CMS. G1 divides the heap into smaller regions and performs garbage collection on a subset of regions during a collection cycle. This approach helps maintain more predictable pause times while also handling compaction.
- **Thread-local handshakes:** Newer GCs like Shenandoah and ZGC leverage thread-local handshakes to minimize STW pauses. By employing this mechanism, they can perform certain GC operations on a per-thread basis, allowing application threads to continue running while the GC works. This approach helps to reduce the overall impact of garbage collection on application performance.
- **Ultra-low-pause-time collectors:** The Shenandoah and ZGC aim to have ultra-low pause times by performing concurrent marking, relocation, and compaction. Both minimize the STW pauses to a small fraction of the overall garbage collection work, offering consistent low latency for applications. While these GCs are not generational in the traditional sense, they do divide the heap into regions and collect different regions at different times. This approach builds upon the principles of incremental and "garbage first" collection. As of this writing, efforts are ongoing to further develop these newer collectors into generational ones, but they are included in this section due to their innovative strategies that enhance the principles of generational garbage collection.

Each collector has its advantages and trade-offs, allowing developers to choose the one that best suits their application requirements.

Young Collections and Weak Generational Hypothesis

In the realm of a generational heap, the majority of allocations take place in the *eden* space of the *young* generation. An allocating thread may encounter an allocation failure when this eden space is near its capacity, indicating that the GC must step in and reclaim space.

During the first *young* collection, the eden space undergoes a scavenging process in which live objects are identified and subsequently moved into the *to* survivor space. The survivor

space serves as a transitional area where surviving objects are copied, aged, and moved back and forth between the *from* and *to* spaces until they cross a tenuring threshold. Once an object crosses this threshold, it is promoted to the *old* generation. The underlying objective here is to promote only those objects that have proven their longevity, thereby creating a "Teenage Wasteland," as Charlie Hunt[5] would explain. 😊

The generational garbage collection is based on two main characteristics related to the weak-generational hypothesis:

1. **Most objects die young:** This means that we promote only long-lived objects. If the generational GC is efficient, we don't promote transients, nor do we promote medium-lived objects. This usually results in smaller long-lived data sets, keeping premature promotions, fragmentation, evacuation failures, and similar degenerative issues at bay.
2. **Maintenance of generations:** The generational algorithm has proven to be a great help to OpenJDK GCs, but it comes with a cost. Because the young-generation collector works separately and more often than the old-generation collector, it ends up moving live data. Therefore, generational GCs incur maintenance/bookkeeping overhead to ensure that they mark all reachable objects—a feat achieved through the use of "write barriers" that track cross-generational references.

Figure 1.1 depicts the three key concepts of generational GCs, providing a visual reinforcement of the information discussed here. The word cloud consists of the following phrases:

- **Objects die young:** Highlighting the idea that most objects are short-lived and only long-lived objects are promoted.
- **Small long-lived data sets:** Emphasizing the efficiency of the generational GC in not promoting transients or medium-lived objects, resulting in smaller long-lived data sets.
- **Maintenance barriers:** Highlighting the overhead and bookkeeping required by generational GCs to mark all reachable objects, achieved through the use of write barriers.

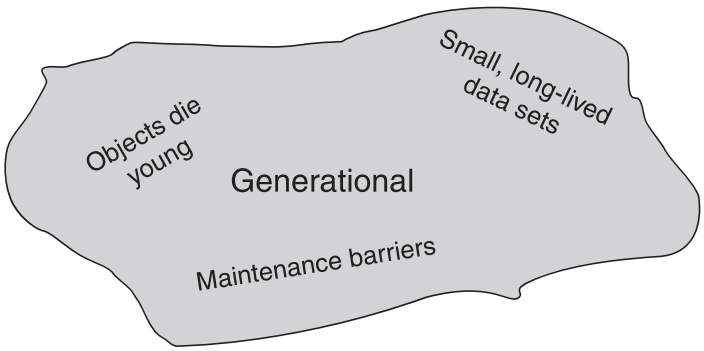

Figure 1.1 Key Concepts for Generational Garbage Collectors

[5] Charlie Hunt is my mentor, the author of *Java Performance* (https://ptgmedia.pearsoncmg.com/images/9780137142521/samplepages/0137142528.pdf), and my co-author for *Java Performance Companion* (www.pearson.com/en-us/subject-catalog/p/java-performance-companion/P200000009127/9780133796827).

Most HotSpot GCs employ the renowned "scavenge" algorithm for young collections. The Serial GC in HotSpot VM employs a single garbage collection thread dedicated to efficiently reclaiming memory within the young-generation space. In contrast, generational collectors such as the Parallel GC (throughput collector), G1 GC, and CMS GC leverage multiple GC worker threads.

Old-Generation Collection and Reclamation Triggers

Old-generation reclamation algorithms in HotSpot VM's generational GCs are optimized for throughput, responsiveness, or a combination of both. The Serial GC employs a single-threaded mark–sweep–compacting (MSC) GC. The Parallel GC uses a similar MSC GC with multiple threads. The CMS GC performs mostly concurrent marking, dividing the process into STW or concurrent phases. After marking, CMS reclaims old-generation space by performing in-place deallocation without compaction. If fragmentation occurs, CMS falls back to the serial MSC.

G1 GC, introduced in Java 7 update 4 and refined over time, is the first incremental collector. Specifically, it incrementally reclaims and compacts the old-generation space, as opposed to performing the single monolithic reclamation and compaction that is part of MSC. G1 GC divides the heap into smaller regions and performs garbage collection on a subset of regions during a collection cycle, which helps maintain more predictable pause times while also handling compaction.

After multiple young-generation collections, the old generation starts filling up, and garbage collection kicks in to reclaim space in the old generation. To do so, a full heap marking cycle must be triggered by either (1) a promotion failure, (2) a promotion of a regular-sized object that makes the old generation or the total heap cross the marking threshold, or (3) a large object allocation (also known as humongous allocation in the G1 GC) that causes the heap occupancy to cross a predetermined threshold.

Shenandoah GC and ZGC—introduced in JDK 12 and JDK 11, respectively—are ultra-low-pause-time collectors that aim to minimize STW pauses. In JDK 17, they are single-generational collectors. Apart from utilizing thread-local handshakes, these collectors know how to optimize for low-pause scenarios either by employing the application threads to help out or by asking the application threads to back off. This GC technique is known as graceful degradation.

Parallel GC Threads, Concurrent GC Threads, and Their Configuration

In the HotSpot VM, the total number of GC worker threads (also known as parallel GC threads) is calculated as a fraction of the total number of processing cores available to the Java process at start-up. Users can adjust the parallel GC thread count by assigning it directly on the command line using the `-XX:ParallelGCThreads=<n>` flag.

This configuration flag enables developers to define the number of parallel GC threads for GC algorithms that use parallel collection phases. It is particularly useful for tuning generational GCs, such as the Parallel GC and G1 GC. Recent additions like Shenandoah and ZGC, also use multiple GC worker threads and perform garbage collection concurrently with the application threads to minimize pause times. They benefit from load balancing, work sharing, and work stealing, which enhance performance and efficiency by parallelizing the garbage collection

process. This parallelization is particularly beneficial for applications running on multi-core processors, as it allows the GC to make better use of the available hardware resources.

In a similar vein, the `-XX:ConcGCThreads=<n>` configuration flag allows developers to specify the number of concurrent GC threads for specific GC algorithms that use concurrent collection phases. This flag is particularly useful for tuning GCs like G1, which performs concurrent work during marking, and Shenandoah and ZGC, which aim to minimize STW pauses by executing concurrent marking, relocation, and compaction.

By default, the number of parallel GC threads is automatically calculated based on the available CPU cores. Concurrent GC threads usually default to one-fourth of the parallel GC threads. However, developers may want to adjust the number of parallel or concurrent GC threads to better align with their application's performance requirements and available hardware resources.

Increasing the number of parallel GC threads can help improve overall GC throughput, as more threads work simultaneously on the parallel phases of this process. This increase may result in shorter GC pause times and potentially higher application throughput, but developers should be cautious not to over-commit processing resources.

By comparison, increasing the number of concurrent GC threads can enhance overall GC performance and expedite the GC cycle, as more threads work simultaneously on the concurrent phases of this process. However, this increase may come at the cost of higher CPU utilization and competition with application threads for CPU resources.

Conversely, reducing the number of parallel or concurrent GC threads may lower CPU utilization but could result in longer GC pause times, potentially affecting application performance and responsiveness. In some cases, if the concurrent collector is unable to keep up with the rate at which the application allocates objects (a situation referred to as the GC "losing the race"), it may lead to a graceful degradation—that is, the GC falls back to a less optimal but more reliable mode of operation, such as a STW collection mode, or might employ strategies like throttling the application's allocation rate to prevent it from overloading the collector.

Figure 1.2 shows the key concepts as a word cloud related to GC work:

- **Task queues:** Highlighting the mechanisms used by GCs to manage and distribute work among the GC threads.
- **Concurrent work:** Emphasizing the operations performed by the GC simultaneously with the application threads, aiming to minimize pause times.
- **Graceful degradation:** Referring to the GC's ability to switch to a less optimal but more reliable mode of operation when it can't keep up with the application's object allocation rate.
- **Pauses:** Highlighting the STW events during which application threads are halted to allow the GC to perform certain tasks.
- **Task stealing:** Emphasizing the strategy employed by some GCs in which idle GC threads "steal" tasks from the work queues of busier threads to ensure efficient load balancing.
- **Lots of threads:** Highlighting the use of multiple threads by GCs to parallelize the garbage collection process and improve throughput.

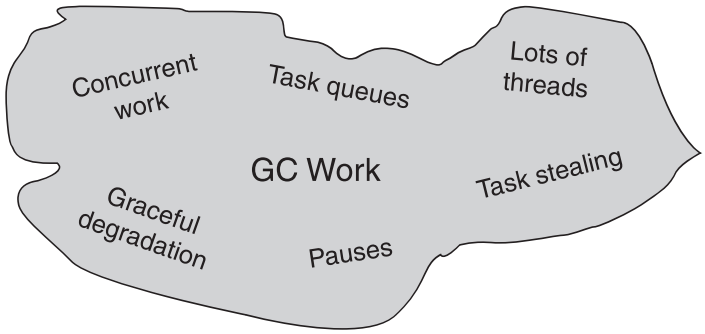

Figure 1.2 Key Concepts for Garbage Collection Work

It is crucial to find the right balance between the number of parallel and concurrent GC threads and application performance. Developers should conduct performance testing and monitoring to identify the optimal configuration for their specific use case. When tuning these threads, consider factors such as the number of available CPU cores, the nature of the application workload, and the desired balance between garbage collection throughput and application responsiveness.

The Evolution of the Java Programming Language and Its Ecosystem: A Closer Look

The Java language has evolved steadily since the early Java 1.0 days. To appreciate the advancements in the JVM, and particularly the HotSpot VM, it's crucial to understand the evolution of the Java programming language and its ecosystem. Gaining insight into how language features, libraries, frameworks, and tools have shaped and influenced the JVM's performance optimizations and garbage collection strategies will help us grasp the broader context.

Java 1.1 to Java 1.4.2 (J2SE 1.4.2)

Java 1.1, originally known as JDK 1.1, introduced JavaBeans, which allowed multiple objects to be encapsulated in a bean. This version also brought Java Database Connectivity (JDBC), Remote Method Invocation (RMI), and inner classes. These features set the stage for more complex applications, which in turn demanded improved JVM performance and garbage collection strategies.

From Java 1.2 to Java 5.0, the releases were renamed to include the version name, resulting in names like J2SE (Platform, Standard Edition). The renaming helped differentiate between Java 2 Micro Edition (J2ME) and Java 2 Enterprise Edition (J2EE).[6] J2SE 1.2 introduced two significant improvements to Java: the Collections Framework and the JIT compiler. The Collections Framework provided "a unified architecture for representing and manipulating collections",[7]

[6] www.oracle.com/java/technologies/javase/javanaming.html

[7] Collections Framework Overview. https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html.

which became essential for managing large-scale data structures and optimizing memory management in the JVM.

Java 1.3 (J2SE 1.3) added new APIs to the Collections Framework, introduced Math classes, and made the HotSpot VM the default Java VM. A directory services API was included for Java RMI to look up any directory or name service. These enhancements further influenced JVM efficiency by enabling more memory-efficient data management and interaction patterns.

The introduction of the New Input/Output (NIO) API in Java 1.4 (J2SE 1.4), based on Java Specification Request (JSR) #51,[8] significantly improved I/O operation efficiency. This enhancement resulted in reduced waiting times for I/O tasks and an overall boost in JVM performance. J2SE 1.4 also introduced the Logging API, which allowed for generating text or XML-formatted log messages that could be directed to a file or console.

J2SE 1.4.1 was soon superseded by J2SE 1.4.2, which included numerous performance enhancements in HotSpot's client and server compilers. Security enhancements were added as well, and Java users were introduced to Java updates via the Java Plug-in Control Panel Update tab. With the continuous improvements in the Java language and its ecosystem, JVM performance strategies evolved to accommodate increasingly more complex and resource-demanding applications.

Java 5 (J2SE 5.0)

The Java language made its first significant leap toward language refinement with the release of Java 5.0. This version introduced several key features, including generics, autoboxing/unboxing, annotations, and an enhanced `for` loop.

Language Features

Generics introduced two major changes: (1) a change in syntax and (2) modifications to the core API. Generics allow you to reuse your code for different data types, meaning you can write just a single class—there is no need to rewrite it for different inputs.

To compile the generics-enriched Java 5.0 code, you would need to use the Java compiler `javac`, which was packaged with the Java 5.0 JDK. (Any version prior to Java 5.0 did not have the core API changes.) The new Java compiler would produce errors if any type safety violations were detected at compile time. Hence, generics introduced type safety into Java. Also, generics eliminated the need for explicit casting, as casting became implicit.

Here's an example of how to create a generic class named `FreshmenAdmissions` in Java 5.0:

```
class FreshmenAdmissions<K, V> {
     //…
}
```

[8] https://jcp.org/en/jsr/detail?id=51

In this example, K and V are placeholders for the actual types of objects. The class `FreshmenAdmissions` is a generic type. If we declare an instance of this generic type without specifying the actual types for K and V, then it is considered to be a raw type of the generic type `FreshmenAdmissions<K, V>`. Raw types exist for generic types and are used when the specific type parameters are unknown.

```
FreshmenAdmissions applicationStatus;
```

However, suppose we declare the instance with actual types:

```
FreshmenAdmissions<String, Boolean>
```

Then `applicationStatus` is said to be a parameterized type—specifically, it is parameterized over types `String` and `Boolean`.

```
FreshmenAdmissions<String, Boolean> applicationStatus;
```

NOTE Many C++ developers may see the angle brackets < > and immediately draw parallels to C++ templates. Although both C++ and Java use generic types, C++ templates are more like a compile-time mechanism, where the generic type is replaced by the actual type by the C++ compiler, offering robust type safety.

While discussing generics, we should also touch on autoboxing and unboxing. In the `FreshmenAdmissions<K, V>` class, we can have a generic method that returns the type V:

```
public V getApprovalInfo() {
     return boolOrNumValue;
}
```

Based on our declaration of V in the parameterized type, we can perform a `Boolean` check and the code would compile correctly. For example:

```
applicationStatus = new FreshmenAdmissions<>();
if (applicationStatus.getApprovalInfo()) {
    //...
}
```

In this example, we see a generic type invocation of V as a `Boolean`. Autoboxing ensures that this code will compile correctly. In contrast, if we had done the generic type invocation of V as an `Integer`, we would get an "incompatible types" error. Thus, autoboxing is a Java compiler conversion that understands the relationship between a primitive and its object class. Just as autoboxing encapsulates a `boolean` value into its `Boolean` wrapper class, unboxing helps return a `boolean` value when the return type is a `Boolean`.

Here's the entire example (using Java 5.0):

```
class FreshmenAdmissions<K, V> {
    private K key;
    private V boolOrNumValue;

    public void admissionInformation(K name, V value) {
        key = name;
        boolOrNumValue = value;
    }

    public V getApprovalInfo() {
        return boolOrNumValue;
    }

    public K getApprovedName() {
        return key;
    }
}

public class GenericExample {

    public static void main(String[] args) {
        FreshmenAdmissions<String, Boolean> applicationStatus;
        applicationStatus = new FreshmenAdmissions<String, Boolean>();

        FreshmenAdmissions<String, Integer> applicantRollNumber;
        applicantRollNumber = new FreshmenAdmissions<String, Boolean>();

        applicationStatus.admissionInformation("Annika", true);

        if (applicationStatus.getApprovalInfo()) {
                applicantRollNumber.admissionInformation(applicationStatus.getApprovedName(), 4);
        }

        System.out.println("Applicant " + applicantRollNumber.getApprovedName() +
            " has been admitted with roll number of " + applicantRollNumber.getApprovalInfo());
    }

}
```

Figure 1.3 shows a class diagram to help visualize the relationship between the classes. In the diagram, the `FreshmenAdmissions` class has three methods: `admissionInformation(K,V)`, `getApprovalInfo():V`, and `getApprovedName():K`. The `GenericExample` class uses the `FreshmenAdmissions` class and has a `main(String[] args)` method.

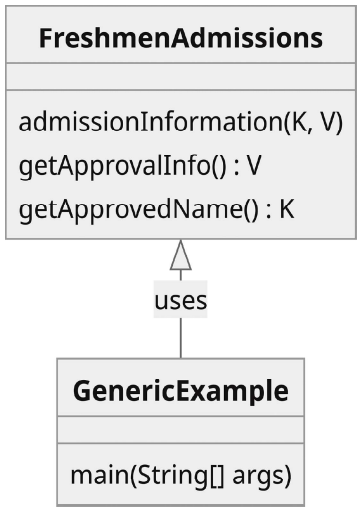

Figure 1.3 A Generic Type Example Showcasing the FreshmenAdmission Class

JVM and Packages Enhancements

Java 5.0 also brought significant additions to the `java.lang.*` and `java.util.*` packages and added most of the concurrent utilities as specified in JSR 166.[9]

Garbage collection *ergonomics* was a term coined in J2SE 5.0; it referred to a default collector for server-class machines.[10] The default collector was chosen to be the parallel GC, and default values for the initial and maximum heap sizes for parallel GC were set automatically.

> **NOTE** The Parallel GC of J2SE 5.0 days didn't have parallel compaction of the old generation. Thus, only the young generation could parallel scavenge; the old generation would still employ the serial GC algorithm, MSC.

Java 5.0 also introduced the enhanced `for` loop, often called the `for-each` loop, which greatly simplified iterating over arrays and collections. This new loop syntax automatically handled the iteration without the need for explicit index manipulation or calls to iterator methods. The enhanced `for` loop was both more concise and less error-prone than the traditional `for` loop, making it a valuable addition to the language. Here's an example to demonstrate its usage:

```
List<String> names = Arrays.asList("Monica", "Ben", "Annika", “Bodin”);
for (String name : names) {
    System.out.println(name);
}
```

[9] Java Community Process. *JSRs: Java Specification Requests*, detail JSR# 166. https://jcp.org/en/jsr/detail?id=166.

[10] https://docs.oracle.com/javase/8/docs/technotes/guides/vm/server-class.html

In this example, the enhanced `for` loop iterates over the elements of the `names` list and assigns each element to the `name` variable in turn. This is much cleaner and more readable than using an index-based `for` loop or an iterator.

Java 5.0 brought several enhancements to the `java.lang` package, including the introduction of annotations, which allows for adding metadata to Java source code. Annotations provide a way to attach information to code elements such as classes, methods, and fields. They can be used by the Java compiler, runtime environment, or various development tools to generate additional code, enforce coding rules, or aid in debugging. The `java.lang.annotation` package contains the necessary classes and interfaces to define and process annotations. Some commonly used built-in annotations are `@Override`, `@Deprecated`, and `@SuppressWarnings`.

Another significant addition to the `java.lang` package was the `java.lang.instrument` package, which made it possible for Java agents to modify running programs on the JVM. The new services enabled developers to monitor and manage the execution of Java applications, providing insights into the behavior and performance of the code.

Java 6 (Java SE 6)

Java SE 6, also known as Mustang, primarily focused on additions and enhancements to the Java API,[11] but made only minor adjustments to the language itself. In particular, it introduced a few more interfaces to the Collections Framework and improved the JVM. For more information on Java 6, refer to the driver JSR 270.[12]

Parallel compaction, introduced in Java 6, allowed for the use of multiple GC worker threads to perform compaction of the old generation in the generational Java heap. This feature significantly improved the performance of Java applications by reducing the time taken for garbage collection. The performance implications of this feature are profound, as it allows for more efficient memory management, leading to improved scalability and responsiveness of Java applications.

JVM Enhancements

Java SE 6 made significant strides in improving scripting language support. The `javax.script` package was introduced, enabling developers to embed and execute scripting languages within Java applications. Here's an example of executing JavaScript code using the `ScriptEngine` API:

```
import javax.script.ScriptEngine;
import javax.script.ScriptEngineManager;
import javax.script.ScriptException;

public class ScriptingExample {
    public static void main(String[] args) {
```

[11] Oracle. "Java SE 6 Features and Enhancements." www.oracle.com/java/technologies/javase/features.html.

[12] Java Community Process. *JSRs: Java Specification Requests*, detail JSR# 270. www.jcp.org/en/jsr/detail?id=270.

```
        FreshmenAdmissions<String, Integer> applicantGPA;
        applicantGPA = new FreshmenAdmissions<String,Integer>();
        applicantGPA.admissionInformation("Annika", 98);

        ScriptEngineManager manager = new ScriptEngineManager();
        ScriptEngine engine = manager.getEngineByName("JavaScript");

        try {
            engine.put("score", applicantGPA.getApprovalInfo());
            engine.eval("var gpa = score / 25; print('Applicant GPA: ' + gpa);");
        } catch (ScriptException e) {
            e.printStackTrace();
        }
    }
}
```

Java SE 6 also introduced the Java Compiler API (JSR 199[13]), which allows developers to invoke the Java compiler programmatically. Using the `FreshmenAdmissions` example, we can compile our application's Java source file as follows:

```
import javax.tools.JavaCompiler;
import javax.tools.ToolProvider;
import java.io.File;

public class CompilerExample {
    public static void main(String[] args) {
        JavaCompiler compiler = ToolProvider.getSystemJavaCompiler();

        int result = compiler.run(null, null, null, "path/to/FreshmenAdmissions.java");
        if (result == 0) {
            System.out.println("Congratulations! You have successfully compiled your program");
        } else {
            System.out.println("Apologies, your program failed to compile");
        }
    }
}
```

Figure 1.4 is a visual representation of the two classes and their relationships.

[13] https://jcp.org/en/jsr/detail?id=199

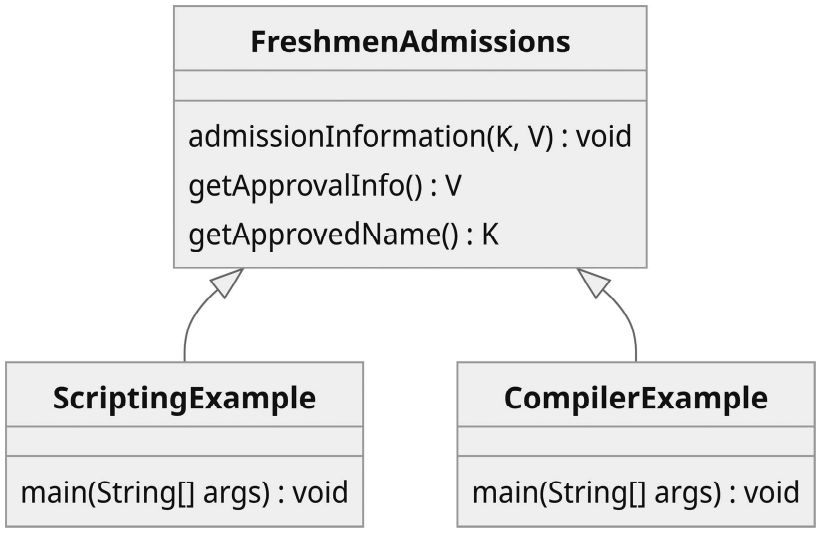

Figure 1.4 Script Engine API and Java Compiler API Examples

Additionally, Java SE 6 enhanced web services support by adding JAXB 2.0 and JAX-WS 2.0 APIs, simplifying the creation and consumption of SOAP-based web services.

Java 7 (Java SE 7)

Java SE 7 marked another significant milestone in Java's evolution, introducing numerous enhancements to both the Java language and the JVM. These enhancements are essential for understanding modern Java development.

Language Features

Java SE 7 introduced the diamond operator (<>), which simplified generic type inference. This operator enhances code readability and reduces redundancy. For instance, when declaring a `FreshmenAdmissions` instance, you no longer need to specify the type parameters twice:

```
// Before Java SE 7
FreshmenAdmissions<String, Boolean> applicationStatus = new FreshmenAdmissions<String, Boolean>();

// With Java SE 7
FreshmenAdmissions<String, Boolean> applicationStatus = new FreshmenAdmissions<>();
```

Another significant addition is the try-with-resources statement, which automates resource management, reducing the likelihood of resource leaks. Resources that implement `java.lang.AutoCloseable` can be used with this statement, and they are automatically closed when no longer needed.

Project Coin,[14] a part of Java SE 7, introduced several small but impactful enhancements, including improving literals, support for strings in switch statements, and enhanced error handling with try-with-resources and multi-catch blocks. Performance improvements such as compressed ordinary object pointers (oops), escape analysis, and tiered compilation also emerged during this time.

JVM Enhancements

Java SE 7 further enhanced the Java NIO library with NIO.2, providing better support for file I/O. Additionally, the fork/join framework, originating from JSR 166: *Concurrency Utilities*,[15] was incorporated into Java SE 7's concurrent utilities. This powerful framework enables efficient parallel processing of tasks by recursively breaking them down into smaller, more manageable subtasks and then merging the outcomes for a result.

The fork/join framework in Java reminds me of a programming competition I participated in during my student years. The event was held by the Institute of Electrical and Electronics Engineers (IEEE) and was a one-day challenge for students in computer science and engineering. My classmate and I decided to tackle the competition's two puzzles individually. I took on a puzzle that required optimizing a binary split-and-sort algorithm. After some focused work, I managed to significantly refine the algorithm, resulting in one of the most efficient solutions presented at the competition.

This experience was a practical demonstration of the power of binary splitting and sorting—a concept that is akin to the core principle of Java's fork/join framework. In the following fork/join example, you can see how the framework is a powerful tool to efficiently process tasks by dividing them into smaller subtasks and subsequently unifying them.

Here's an example of using the fork/join framework to calculate the average GPA of all admitted students:

```
// Create a list of admitted students with their GPAs
List<Integer> admittedStudentGPAs = Arrays.asList(95, 88, 76, 93, 84, 91);

// Calculate the average GPA using the fork/join framework
ForkJoinPool forkJoinPool = new ForkJoinPool();
AverageGPATask averageGPATask = new AverageGPATask(admittedStudentGPAs, 0,
                                                   admittedStudentGPAs.size());
double averageGPA = forkJoinPool.invoke(averageGPATask);

System.out.println("The average GPA of admitted students is: " + averageGPA);
```

In this example, we define a custom class `AverageGPATask` extending `RecursiveTask<Double>`. It's engineered to compute the average GPA from a subset of students by recursively splitting the list of GPAs into smaller tasks—a method reminiscent of the divide-and-conquer approach I refined during my competition days. Once the tasks are

[14] https://openjdk.org/projects/coin/

[15] https://jcp.org/en/jsr/detail?id=166