Simi Docs / Java

Source: https://github.com/saidake/simi-docs/tree/release/simi-docs-1.6.3

Edited by: Craig Brown
Version: 1.4.2
Date: May 13, 2025

Java / Concept

Java Environment

JDK (Java Development Kit)

Definition

JDK stands for Java Development Kit. It's a software development environment that's used to create Java applications and applets.

Components

- Java Virtual Machine (JVM)
- Class libraries

- Java Runtime Environment (JRE)

- Compiler (javac)
- Interpreter (java)
- Archiver (jar)
- Documentation generator (javadoc)

Versions

- JDK 1.0 (January 1996)
- JDK 1.1 (February 1997)
- J2SE 1.2 (December 1998)
- J2SE 1.3 (May 2000)
- J2SE 1.4 (February 2002)
- J2SE 5.0 (September 2004) - Also known as JDK 5.0 (note the version jump)
- Java SE 6 (December 2006) - Originally called J2SE 6.0
- Java SE 7 (July 2011)
- Java SE 8 (March 2014)
- Java SE 9 (September 2017)
- Java SE 10 (March 2018)
- Java SE 11 (September 2018) - Long Term Support (LTS)
- Java SE 12 (March 2019)
- Java SE 13 (September 2019)
- Java SE 14 (March 2020)
- Java SE 15 (September 2020)
- Java SE 16 (March 2021)
- Java SE 17 (September 2021) - Long Term Support (LTS)
- Java SE 18 (March 2022)
- Java SE 19 (September 2022)
- Java SE 20 (March 2023)
- Java SE 21 (September 2023) - Long Term Support (LTS)

JRE (Java Runtime Environment)

Definition

Ordinary users do not need to install JDK to run Java programs, do not need a compiler, and only need to install JRE.

Components

- Java Virtual Machine (JVM)
- Class libraries

JVM (Java Virtual Machine)

Definition

The Java Virtual Machine is an abstract computing machine.

Like a real computing machine, it has an instruction set and manipulates various memory areas at run time.

Essentially, it is a program. After it starts, it can execute instructions in the bytecode file.

Initialization Time

When a program starts running, the virtual machine begins to be instantiated.

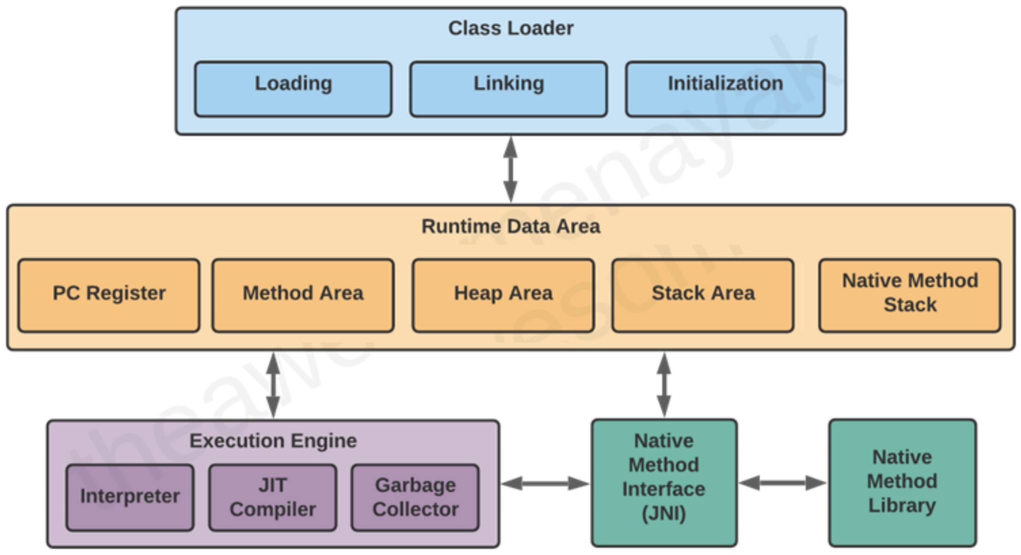
When multiple programs start, there will be multiple virtual machine instances, If the program exits or closes, the virtual machine instance will disappear.

Data Sharing

Data cannot be shared between multiple virtual machine instances.

Components

- Class Loader
- Runtime Data Area
- Execution Engine



Compilation process

1) Java source file -> javac compiler -> bytecode file
2) Bytecode file -> JVM interpreter -> Machine code (for specific machines)

Dynamic compilation (refers to compiling at runtime)

1) Ahead-of-time compilation (AOT) It refers to compiling before running, also known as static compilation.
2) Adaptive dynamic compilation A type of dynamic compilation, but it usually executes later than JIT compilation, allowing the program to run in a certain form first, collecting some information, and then performing dynamic compilation.
3) Just-in-time compilation (JIT) JIT compilation refers to the compilation of a piece of code when it is about to be executed for the first time, hence it is called instant compilation.

JIT compilation is a special case of dynamic compilation.

Compiles the given bytecode instruction sequence to machine code at runtime before executing it natively.

Difference between interpreter and JIT compiler

- An interpreter executes source code directly, while a Just-in-Time (JIT) compiler compiles the most frequently used code while the program is running.
- Interpreter
  - Function:
    - The interpreter reads and executes Java bytecode instructions one at a time. It translates each bytecode instruction into machine code and executes it immediately.
    - the entire process of the interpreter operates at runtime.
  - Execution Process:
    - Direct Execution: The interpreter executes the bytecode directly without compiling it into native machine code beforehand.
    - Step-by-Step: It processes bytecode instructions one by one, which means it translates bytecode to machine code on-the-fly for each instruction as it is encountered.
  - Advantages:
    - Simplicity: The interpreter is simpler and quicker to implement and start up because it doesn’t need to compile bytecode into native code.
    - Flexibility: It allows the JVM to start executing code quickly, which can be useful for quick startup and for small programs or applications with varying code paths.
  - Disadvantages:
    - Performance: Interpretation tends to be slower compared to compiled execution because each bytecode instruction must be translated into machine code each time it is executed.
    - Repeated Work: Frequently executed code paths are interpreted repeatedly, which can result in inefficient execution.
- JIT Compiler
  - Function:
    - The JIT compiler translates Java bytecode into native machine code at runtime. It compiles bytecode into machine code just before execution (hence "Just-In-Time").
    - The entire process of the Just-In-Time (JIT) compiler occurs at runtime.
  - Execution Process:
    - Compilation: The JIT compiler analyzes the bytecode, compiles frequently executed code paths (hot spots) into native machine code, and caches this compiled code.
    - Execution: Once compiled, the native machine code is executed directly by the CPU, bypassing the need for further interpretation of those code paths.
  - Advantages:
    - Performance: JIT compilation generally results in significant performance improvements because the compiled machine code runs directly on the CPU, avoiding the overhead of interpretation.
    - Optimization: The JIT compiler can perform various optimizations based on runtime profiling, such as inlining methods, optimizing loops, and reducing method call overhead.
  - Disadvantages:
    - Startup Time: The JIT compiler introduces a delay in startup time because it needs to compile bytecode before executing it. This delay can be mitigated with warm-up time as frequently executed code gets compiled.
    - Memory Usage: Compiling and caching native code consumes additional memory, which can be a concern in memory-constrained environments.

Portability

- The interpreters for each platform are different, but the virtual machines implemented are the same, which is why Java can cross platforms.

JVM type

1) Free and open source implementations
    - HotSpot JVM (interpreter + JIT compiler)

      HotSpot is a Java virtual machine (JVM) for desktop and server computers.

      It was originally developed by Sun Microsystems and is now maintained and distributed by Oracle Corporation.

      It is a virtual machine included in Oracle JDK and Open JDK, and is currently the most widely used Java virtual machine.

      (Oracle acquired Sun Microsystems in 2010, so Sun JDK is now Oracle JDK.)
    - JamVM (interpreter)
2) Proprietary implementations
    - JRockit VM (interpreter + JIT compiler)

      JRockit is a Java virtual machine (JVM) that allows Java applications to run on Windows and Linux operating systems. It's especially well-suited for running Oracle WebLogic Server.

      JRockit focuses on server-side applications, which can be less focused on program startup speed.

      JRockit does not include an interpreter implementation internally, and all code is compiled and executed by the JIT compiler.

Version Features

Java 8 (March 2014)

Default Methods

Allows methods in interfaces to have a default implementation, enabling the evolution of APIs without breaking existing code.

```
interface Vehicle {
    default void start() {
        System.out.println("Starting the vehicle...");
    }
}
```

Lambda Expressions

Introduced functional programming style by allowing expressions in functional interfaces.

```
Runnable r = () -> System.out.println("Hello, World!");
```

Stream API

Provides a declarative way to process collections of objects with operations like filter, map, and reduce.

```
List<String> names = Arrays.asList("Alice", "Bob", "Charlie");
names.stream().filter(s -> s.startsWith("A")).forEach(System.out::println);
```

Optional Class

Helps avoid NullPointerException by providing a type-safe way to handle null values.

```
Optional<String> name = Optional.ofNullable("Alice");
name.ifPresent(System.out::println);
```

java.time Package (Date and Time API)

A new API for dates, times, instants, and durations, providing improved handling of date and time operations.

```
LocalDate today = LocalDate.now();
LocalDate birthday = LocalDate.of(1990, Month.JANUARY, 1);
```

Base64 API

Provides built-in methods for encoding and decoding Base64.

Nashorn JavaScript Engine

A new engine to run JavaScript code in Java applications.

Java 9 (September 2017)

Module System (Project Jigsaw)

Introduces a module system that encapsulates packages and enforces stronger encapsulation, improving the structure and security of applications.

```
module com.example.module {
    exports com.example.package;
}
```

jshell (Java Shell)

A REPL (Read-Eval-Print Loop) tool that allows interactive execution of Java code for quick experimentation.

Factory Methods for Collections

Static factory methods like List.of(), Set.of(), and Map.of() to create immutable collections more concisely.

```
List<String> names = List.of("Alice", "Bob", "Charlie");
```

Stream API Enhancements

Adds new methods like takeWhile, dropWhile, iterate to improve stream processing.

Private Interface Methods

Allows interfaces to have private methods to encapsulate helper methods.

Java 10 (March 2018)

var for Local Variables

Introduces var keyword for local variable type inference, allowing the compiler to infer the type of the variable.

```
var message = "Hello, Java 10!";
```

Performance Improvements for the G1 Garbage Collector

Reduces Full GC pause times, making it the default garbage collector.

Application Class-Data Sharing (AppCDS)

Improves startup time and reduces footprint by sharing class data across multiple JVMs.

Java 11 (September 2018)

var in Lambda Parameters

Extends var usage to lambda expressions for consistency.

New String Methods

Adds methods like isBlank(), lines(), strip(), repeat().

HTTP Client (Standardized)

Provides an improved and standardized HTTP client API for handling HTTP requests and responses.

```
HttpClient client = HttpClient.newHttpClient();
HttpRequest request = HttpRequest.newBuilder().uri(URI.create("https://example.com")).build();
```

Removal of java.se.ee Module

Removes deprecated modules like java.xml.ws, java.activation, java.corba, etc.

Java 12 (March 2019)

Switch Expressions (Preview)

Introduces switch expressions, allowing switch to return a value, making it more concise and expressive.

```
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY                -> 7;
    default                     -> throw new IllegalStateException("Unexpected value: " + day);
};
```

JVM Constants API

Introduces an API to model key class-file and runtime artifacts, such as constant pool entries.

Java 13 (September 2019)

Switch Expressions (Preview)

This feature extended switch statements to be used as expressions, allowing for a more concise syntax.

The switch expression can return a value and can use the -> (arrow) label or yield to specify the value to return.

```
int result = switch (day) {
    case MONDAY -> 1;
    case TUESDAY -> 2;
    default -> {
        System.out.println("Unknown day");
        yield -1;
        }
    };
```

Text Blocks (Preview)

Introduces text blocks, allowing multi-line string literals that preserve the formatting of the source code.

```
String text = """
    This is a text block.
    It can span multiple lines.
    """;
```

Java 14 (March 2020)

Pattern Matching for instanceof (Preview)

Simplifies the code by combining instanceof checks and casting.

```
if (obj instanceof String s) {
    System.out.println(s.toLowerCase());
}
```

Helpful NullPointerExceptions

Improves the error message by providing more detailed context on NullPointerExceptions.

Java 15 (September 2020)

Sealed Classes (Preview)

Enables developers to restrict which other classes can extend or implement them, enhancing class hierarchy control.

```
public sealed class Shape permits Circle, Square { }

final class Circle extends Shape { }
final class Square extends Shape { }
```

Java 16 (March 2021)

Pattern Matching for instanceof (Standardized)

Finalizes the feature introduced in Java 14.

```
if (obj instanceof String s) {
    System.out.println(s.toLowerCase());
}
```

Records (Standardized)

Provides a compact syntax for declaring classes whose main purpose is to hold data.

Records automatically generate boilerplate code like constructors, equals(), hashCode(), and toString().

```
public record Point(int x, int y) {}
```

Usage

```
public class Main {
    public static void main(String[] args) {
        Person person = new Person("Alice", 25);
        System.out.println(person.name());  // Alice
        System.out.println(person.age());   // 25
        System.out.println(person);         // Person[name=Alice, age=25]
    }
}
```

Adding Custom Methods

```
public record Person(String name, int age) {
    public String greet() {
        return "Hello, my name is " + name;
    }
}
```

Custom Constructor

```
public record Person(String name, int age) {
    public Person {
        if (age < 0) throw new IllegalArgumentException("Age cannot be negative");
    }
}
```

Static Fields and Methods

```
public record Person(String name, int age) {
    static String species = "Homo sapiens";
    public static String getSpecies() {
        return species;
    }
}
```

Records with Collections

```
import java.util.List;
public record Person(String name, List<String> hobbies) { }

public class Main {
    public static void main(String[] args) {
        List<String> hobbies = List.of("Reading", "Coding");
        Person p = new Person("Alice", hobbies);
        System.out.println(p.hobbies()); // [Reading, Coding]
    }
}
```

Using Records with Java Streams

```
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Person> people = List.of(
            new Person("Alice", 25),
            new Person("Bob", 30)
        );

        people.stream()
              .map(Person::name)
              .forEach(System.out::println);
    }
}
```

Vector API (Incubator)

Introduces an API for expressing vector computations that compile efficiently on supported hardware.

Java 17 (Released in 2021, LTS Version)

Sealed Classes

Enables developers to restrict which other classes can extend or implement them, enhancing class hierarchy control.

```
public sealed class Shape permits Circle, Square { }

final class Circle extends Shape { }
final class Square extends Shape { }
```

Pattern Matching for switch (Preview)

Introduces pattern matching capabilities to switch statements.

Pattern Matching in switch allows you to match an object's type and deconstruct it within the same expression.

```
public class PatternMatchingSwitchExample {
    public static void main(String[] args) {
        Object obj = 123; // Try changing this to different values like "Hello", 3.14, or null

        String result = switch (obj) {
            case null -> "The object is null.";
            case Integer i -> "It's an Integer with value: " + i;
            case String s -> "It's a String with length: " + s.length();
            case Double d -> "It's a Double with value: " + d;
            default -> "Unknown type";
        };

        System.out.println(result);
    }
}
```

Text Blocks

Adds support for multi-line string literals, simplifying the writing and maintenance of lengthy strings.

Enhanced RandomGenerator Interface

Improves the randomness generation capabilities.

```
import java.util.random.RandomGenerator;

public class Main {
    public static void main(String[] args) {
        RandomGenerator random = RandomGenerator.of("L64X128MixRandom");
        System.out.println(random.nextInt(100)); // Generates a random integer between 0 and 99
    }
}
```

Context-Specific Deserialization Filters

Enhances security controls for object deserialization processes.

```
import java.io.*;

public class Main {
    public static void main(String[] args) throws Exception {
        ObjectInputFilter filter = ObjectInputFilter.Config.createFilter("java.base/*;!*");
        ObjectInputStream ois = new ObjectInputStream(new FileInputStream("object.dat"));
        ois.setObjectInputFilter(filter);
        // Deserialize object
    }
}
```

Foreign Function & Memory API (Incubator)

Provides safer and more efficient interaction with non-Java code.

Java 19 (Released in 2022)

Virtual Threads (1st Preview, JEP 425):

Introduces lightweight, user-mode threads that dramatically reduce the effort for writing, maintaining, and observing high-throughput concurrent applications.

Structured Concurrency (1st Incubator, JEP 428):

Simplifies multi-threaded programming by handling multiple tasks in a structured manner.

Record Patterns (2nd Preview, JEP 405):

Enhances pattern matching by allowing deconstruction of record values.

Pattern Matching for switch (3rd Preview, JEP 427):

Continues to develop pattern matching capabilities for switch statements, making them more expressive and flexible.

Foreign Function & Memory API (3rd Incubator, JEP 424):

Further refines the API for interfacing with code and data outside of the JVM.

Vector API (4th Incubator, JEP 426):

Continues to evolve the API for vector computations to improve performance on supported hardware.

Java 21 (Released in 2024)

Pattern Matching for switch (5th Preview) - JEP 440:

Extends pattern matching capabilities in switch statements for more complex expressions.

Record Patterns (3rd Preview) - JEP 441:

Enhances record classes for pattern matching in constructs like instanceof and switch.

Unnamed Patterns and Variables - JEP 443:

Introduces unnamed patterns (_) and variables (_) to ignore unused pattern matching results.

Virtual Threads (2nd Preview) - JEP 444:

Improves lightweight threads (virtual threads) for enhanced concurrency performance.

String Templates (1st Preview) - JEP 430:

Adds template strings for easier string interpolation and formatting.

JVM Structure

Reference:

The JVM Run-Time Data Areas

https://www.baeldung.com/java-jvm-run-time-data-areas
https://blog.jamesdbloom.com/JVMInternals.html

Core Components

The PC Register (thread private)

Definition

The PC Register is a runtime data area used by the JVM to hold the address of the current instruction being executed by a thread.

Each thread has its own PC register, and the PC register is updated with the next instruction after it is executed.

Creation Timing

A PC Register is created every time a new thread is created.

Content in PC register

At any point, each Java Virtual Machine thread is executing the code of a single method, namely the current method (§2.6) for that thread.

- If that method is not native, the pc register contains the address of the Java Virtual Machine instruction currently being executed.
- If the method currently being executed by the thread is native, the value of the Java Virtual Machine's pc register is undefined.

Exception

The Java Virtual Machine's pc register is wide enough to hold a returnAddress or a native pointer on the specific platform.

Method Area (thread shared)

Definition

The Method Area is a shared data area in the JVM that stores class and interface structure data.

Creation Time

It is created when the JVM starts, and it is destroyed only when the JVM exits.

Contents of the Method Area

It stores the class and interface structure data loaded by the Class Loader,

Note: Please refer to the ClassFile Structure.

Here are some examples of class and interface structure data that is stored in the Method Area:

- The fully qualified name of the java.lang.Object class
- The fully qualified name of the java.lang.String class
- The modifiers for the java.lang.Object class
- The fields declared by the java.lang.Object class
- The methods declared by the java.lang.Object class
- The constructors declared by the java.lang.Object class
- The run-time constant pool for the java.lang.Object class

Implementation

Although the method area is logically part of the heap, simple implementations may choose not to either garbage collect or compact it.

This specification does not mandate the location of the method area or the policies used to manage compiled code.

Difference between metaspace and method area

Method Area is also known as the Permanent Generation. Starting from Java SE 8, the Permanent Generation is replaced by the Metaspace, which is located in native memory.

- Fixed Size Issues:

    PermGen

    The PermGen space had a fixed maximum size, which could lead to OutOfMemoryError if the space

was exhausted.
This was problematic for applications with a large number of classes or heavy use of reflection, which could dynamically generate and load classes.

Metaspace

Metaspace, on the other hand, is allocated in native memory and can grow dynamically as needed, limited only by the amount of available system memory.
This removes the need to specify a maximum size and reduces the likelihood of OutOfMemoryError due to PermGen space exhaustion.

- Simplified Memory Management:

  PermGen

  Managing the PermGen space was complex and often led to memory leaks, particularly with frameworks that dynamically generated classes (e.g., Hibernate, Spring).
  Classes loaded by different class loaders could remain in memory longer than necessary, causing the PermGen space to fill up.

  Metaspace

  Metaspace simplifies memory management by allocating class metadata in native memory, which is managed by the operating system.
  This change helps reduce the risk of memory leaks and simplifies garbage collection, as the JVM no longer needs to manage the class metadata in the Java heap.

- Performance Improvements:

  PermGen

  Garbage collection in PermGen was less efficient and could negatively impact application performance, especially during full GC cycles.

  Metaspace

  By moving class metadata to native memory, Metaspace allows for more efficient garbage collection and better overall performance.
  This is because class metadata is now managed separately from the Java heap, allowing the heap to be collected and optimized independently.

- Configuration and Tuning

  PermGen

  Developers needed to tune the size of the PermGen space using JVM options (-XX:PermSize and -XX:MaxPermSize), which could be challenging and error-prone.

  Metaspace

  With Metaspace, developers have fewer tuning parameters to worry about, and the JVM can more effectively manage memory usage automatically.
  While there is an option to set a maximum size for Metaspace (-XX:MaxMetaspaceSize), it is not typically necessary to adjust this setting.

Size of the Method Area

The method area may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger method area becomes unnecessary.
The memory for the method area does not need to be contiguous.

A Java Virtual *Machine implementation may provide the programmer or the user control over the initial size of the method area,*
as well as*, in the case of a varying-size method area, control over the maximum and minimum method area size.*

Exception

- If memory in the method area cannot be made available to satisfy an allocation request, the Java Virtual Machine throws an OutOfMemoryError.

Run-Time Constant Pool

1. What's the Run-Time constant Pool?

   A run-time constant pool is a per-class or per-interface run-time representation of the constant_pool table in a class file.
   Each run-time constant pool is allocated from the Java Virtual Machine's Method Area (§2.5.4).
   The constant_pool table (§4.4) in the binary representation of a class or interface is used to construct the run-time constant pool upon class or interface creation (§5.3).
   All references in the run-time constant pool are initially symbolic.
   Note: Please refer to the ClassFile Structure.

   The symbolic references in the run-time constant pool are derived from structures in the binary representation of the class or interface as follows:

   - A symbolic reference to a class or interface is derived from a CONSTANT_Class_info structure (§4.4.1) in the binary representation of a class or interface.
     Such a reference gives the name of the class or interface in the form returned by the Class.getName method, that is:
     - For a nonarray class or an interface, the name is the binary name (§4.2.1) of the class or interface.
     - For an array class of *n* dimensions, the name begins with *n* occurrences of the ASCII "[" character followed by a representation of the element type:
       - If the element type is a primitive type, it is represented by the corresponding field descriptor (§4.3.2).
       - Otherwise, if the element type is a reference type, it is represented by the ASCII "L" character followed by the binary name (§4.2.1) of the element type followed by the ASCII ";" character.

     Whenever this chapter refers to the name of a class or interface, it should be understood to be in the form returned by the Class.getName method.
   - A symbolic reference to a field of a class or an interface is derived from a CONSTANT_Fieldref_info structure (§4.4.2) in the binary representation of a class or interface.
     Such a reference gives the name and descriptor of the field, as well as a symbolic reference to the class or interface in which the field is to be found.
   - A symbolic reference to a method of a class is derived from a CONSTANT_Methodref_info structure (§4.4.2) in the binary representation of a class or interface.
     Such a reference gives the name and descriptor of the method, as well as a symbolic reference to the class in which the method is to be found.
   - A symbolic reference to a method of an interface is derived from a CONSTANT_InterfaceMethodref_info structure (§4.4.2) in the binary representation of a class or interface.
     Such a reference gives the name and descriptor of the interface method, as well as a symbolic reference to the interface in which the method is to be found.
   - A symbolic reference to a method handle is derived from a CONSTANT_MethodHandle_info structure (§4.4.8) in the binary representation of a class or interface.
   - A symbolic reference to a method type is derived from a CONSTANT_MethodType_info structure (§4.4.9) in the binary representation of a class or interface.
   - A symbolic reference to a *call site specifier* is derived from a CONSTANT_InvokeDynamic_info structure (§4.4.10) in the binary representation of a class or interface.
     Such a reference gives:
     - a symbolic reference to a method handle, which will serve as a bootstrap method for an *invokedynamic* instruction (§*invokedynamic*);

        - a sequence of symbolic references (to classes, method types, and method handles), string literals, and run-time constant values which will serve as *static arguments* to a bootstrap method;
        - a method name and method descriptor.

    In addition, certain run-time values which are not symbolic references are derived from items found in the constant_pool table:

- A string literal is a reference to an instance of class String, and is derived from a CONSTANT_String_info structure (§4.4.3) in the binary representation of a class or interface.

  The CONSTANT_String_info structure gives the sequence of Unicode code points constituting the string literal.

  The Java programming language requires that identical string literals (that is, literals that contain the same sequence of code points) must refer to the same instance of class String (JLS §3.10.5). In addition, if the method String.intern is called on any string, the result is a reference to the same class instance that would be returned if that string appeared as a literal. Thus, the following expression must have the value true:

  ```
  ("a" + "b" + "c").intern() == "abc"
  ```

  To derive a string literal, the Java Virtual Machine examines the sequence of code points given by the CONSTANT_String_info structure.
    - If the method String.intern has previously been called on an instance of class String containing a sequence of Unicode code points identical to that given by the CONSTANT_String_info structure, then the result of string literal derivation is a reference to that same instance of class String.
    - Otherwise, a new instance of class String is created containing the sequence of Unicode code points given by the CONSTANT_String_info structure; a reference to that class instance is the result of string literal derivation. Finally, the intern method of the new String instance is invoked.
- Run-time constant values are derived from CONSTANT_Integer_info, CONSTANT_Float_info, CONSTANT_Long_info, or CONSTANT_Double_info structures (§4.4.4, §4.4.5) in the binary representation of a class or interface.

  Note that CONSTANT_Float_info structures represent values in IEEE 754 single format and CONSTANT_Double_info structures represent values in IEEE 754 double format (§4.4.4, §4.4.5). The run-time constant values derived from these structures must thus be values that can be represented using IEEE 754 single and double formats, respectively.

The remaining structures in the constant_pool table of the binary representation of a class or interface - the CONSTANT_NameAndType_info and CONSTANT_Utf8_info structures (§4.4.6, §4.4.7) - are only used indirectly when deriving symbolic references to classes, interfaces, methods, fields, method types, and method handles, and when deriving string literals and call site specifiers.

2. When is the Run-Time Constant Pool constructed?

   The run-time constant pool for a class or interface is constructed when the class or interface is created (§5.3) by the Java Virtual Machine.

3. What is contained in the Run-Time Constant Pool?

   It contains several kinds of constants, ranging from numeric literals known at compile-time to method and field references that must be resolved at run-time.

   (symbolic references to the class and interface names, field names, and method names)

   the runtime constant pool in Java contains symbolic references, not actual references.

   Note: Please refer to the Constant Pool.

   Resolution at Run-Time:

      Resolution Process:

During the execution of a Java program, the JVM needs to resolve method and field references to actual methods and fields in memory. This process involves:

Method Resolution:

Finding the actual method implementation in the class or its superclasses and ensuring the method's signature matches the reference.

Field Resolution:

Finding the actual field in the class or its superclasses and ensuring that the field's type and name match the reference.

Dynamic Resolution:

This resolution is dynamic because it occurs at runtime, not at compile time. The JVM uses the constant pool to look up and resolve these references as the program executes.

1. The following exceptional condition is associated with the construction of the run-time constant pool for a class or interface:
    - When creating a class or interface, if the construction of the run-time constant pool requires more memory than can be made available in the method area of the Java Virtual Machine,

      the Java Virtual Machine throws an OutOfMemoryError.

      See §5 (Loading, Linking, and Initializing) for information about the construction of the run-time constant pool.

Heap (thread shared)

Definition

The Heap is a runtime data area where all Java objects (all class instances and arrays) are stored (Arrays and object references are stored in the Java Stack).

Thus, whenever we create a new class instance or array, the JVM will find some available memory in the Heap and assign it to the object.

Creation Time

The Heap's creation occurs at the JVM start-up, and its destruction happens at the exit.

Contents in Java Heap?

All Java objects (all class instances and arrays)

Java Heap algorithm

Heap storage for objects is reclaimed by an automatic storage management system (known as a garbage collector); objects are never explicitly deallocated. The Java Virtual Machine assumes no particular type of automatic storage management system,

and the storage management technique may be chosen according to the implementor's system requirements.

String Pool

1) What is String Pool?

   The Java String Pool is a storage area in the Java heap that stores string literals.

   Shared String objects are maintained in the string pool, and these strings will not be recycled by the garbage collector.

   It's also known as the String Intern Pool or String Constant Pool.

   String Pool is possible because String is immutable in Java and it is an implementation of String interning concept.

2) What's the process of creating string?

   When we use double quotes to create a String, it first looks for String with the same value in the String pool, if found it just returns the reference, otherwise it creates a new String in the pool and then returns the reference.

String Pool Object

```
String str = "Cat";
```

The "Cat" in the String Pool is indeed a java.lang.String object.
The total number of string objects created is 1.
If there is already a string literal "Cat" in the pool, then the "Cat" object will be reused.
Otherwise, a "Cat" object will be created in the pool.

java.lang.String Object

```
String str = new String("Cat");
```

In the above statement, either 1 or 2 string objects will be created.
If there is already a string literal "Cat" in the pool, then:

- A "Cat" object in the pool will be reused.
- A new String object with the value "Cat" will be created in the heap space, so a total of 2 string objects will be created.

If there is no string literal "Cat" in the pool, then:

- A "Cat" object will be created in the pool.
- A new String object with the value "Cat" will be created in the heap space, so a total of 2 string objects will be created.

Difference between String Pool Object and java.lang.String Object

```
String poolString = "Cat"; // From the String Pool
String newString = new String("Cat"); // New object in heap
System.out.println(poolString == newString); // This will print "false"
```

Although both contain the same string value, they are different objects with different memory locations.

Size

The heap may be of a fixed size or may be expanded as required by the computation and may be contracted if a larger heap becomes unnecessary.
The memory for the heap does not need to be contiguous.

A Java Virtual Machine implementation may provide the programmer or the user control over the initial size of the heap, as well as,
if the heap can be dynamically expanded or contracted, control over the maximum and minimum heap size.

Java Heap Exception

If a computation requires more heap than can be made available by the automatic storage management system, the Java Virtual Machine throws an OutOfMemoryError.

Escape Analysis

Dynamic Scope

Escape Analysis is a method for determining the dynamic scope of objects -- where in the program an object can be accessed.
Escape Analysis determines all the places where an object can be stored and whether the lifetime of the object can be proven to be restricted only to the current method and/or thread.

Process

If the object does not escape, then the JVM could, for example, do something similar to an “automatic stack allocation” of the object.
In this case, the object would not be allocated on the heap and it would never need to be managed by the garbage collector.
As soon as the method containing the stack-allocated object returned, the memory that the object used would immediately be freed.

Escape types

Within the HotSpot VM source code, you can see how the EA analysis system classifies the usage of each object:

typedef enum {
  NoEscape = 1,   // An object does not escape method or thread and it is not passed to call. It could be replaced with scalar.
  ArgEscape = 2,   // An object does not escape method or thread but it is passed as argument to call or referenced by argument and it does not escape during call.
  GlobalEscape = 3   // An object escapes the method or thread.
}

Optimization

1) The first option suggests that the object can be replaced by a scalar substitute.
   This elimination is called scalar replacement. This means that the object is broken up into its component fields,
   which are turned into the equivalent of extra local variables in the method that allocates the object.

   Once this has been done, another HotSpot VM JIT technique can kick in, which enables these object fields (and the actual local variables) to be stored in CPU registers (or on the stack if necessary).
2) Method inlining is one of the first optimizations and is known as a gateway optimization,
   because it opens the door to other techniques by first bringing related code closer together.

```
public class Rect {
   private int w;
   private int h;
   public Rect(int w, int h) {
      this.w = w;
      this.h = h;
   }
   public int area() {
      return w * h;
   }
   public boolean sameArea(Rect other) {
      return this.area() == other.area();
   }
   public static void main(final String[] args) {
      java.util.Random rand = new java.util.Random();
      int sameArea = 0;
      for (int i = 0; i < 100_000_000; i++) {
         Rect r1 = new Rect(rand.nextInt(5), rand.nextInt(5));
         Rect r2 = new Rect(rand.nextInt(5), rand.nextInt(5));
         if (r1.sameArea(r2)) { sameArea++; }
      }
      System.out.println("Same area: " + sameArea);
   }
}

public boolean sameArea(Rect);
   Code:
      0: aload_0
      1: invokevirtual #4    // Method area:()I
      4: aload_1
      5: invokevirtual #4    // Method area:()I
      8: if_icmpne     15
     11: iconst_1
     12: goto          16
     15: iconst_0
     16: ireturn

public int area();
   Code:
```

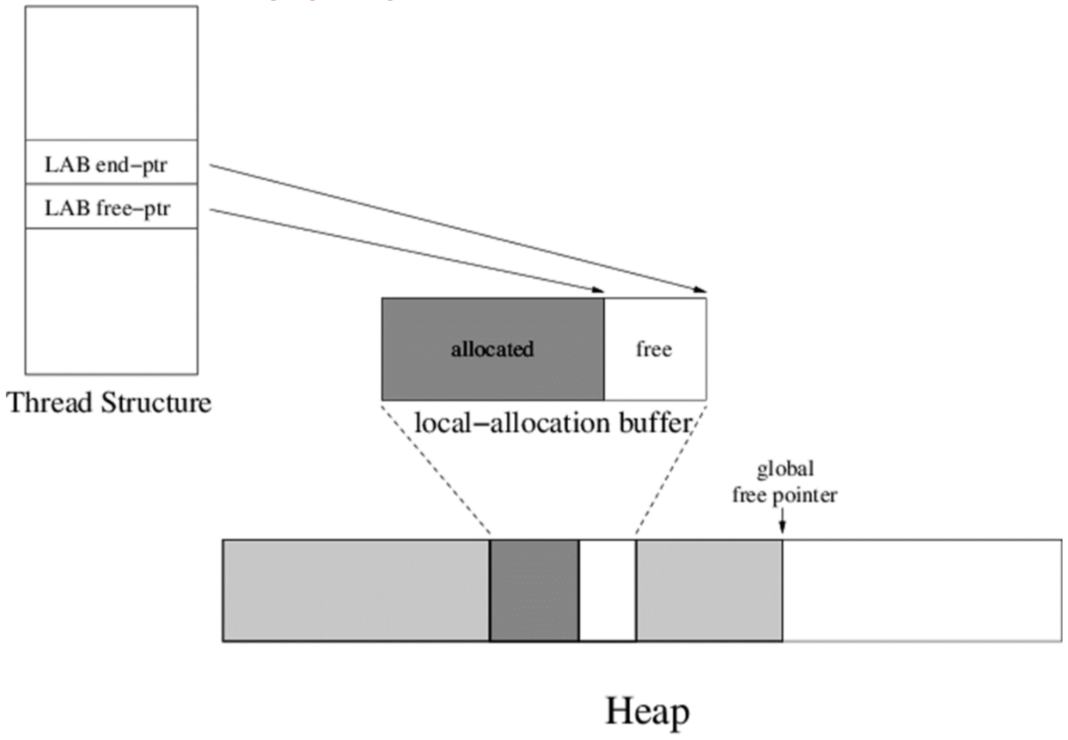
```
0: aload_0       #2    // Field w:I

1: getfield
4: aload_0
5: getfield      #3    // Field h:I

8: imul
9: ireturn
```

Now that the call to sameArea() and the calls to area have been inlined,
the method scopes no longer exist,
and the variables are present only in the scope of main().
This means that EA will no longer treat either r1 or r2 as an ArgEscape:
both are now classified as a NoEscape after the methods have been fully inlined.

Thread Local Allocation Buffer

Location

TLAB stands for Thread Local Allocation Buffer and it is a region inside Eden, which is exclusively assigned to a thread.
In other words, only a single thread can allocate new objects in this area. Each thread has own TLAB.
https://www.researchgate.net/publication/221137885_Supporting_per-processor_local-allocation_buffers_using_lightweight_user-level_preemption_notification



Purpose

Thread-Specific Allocation

Each thread in the JVM has its own TLAB, a small portion of the heap space reserved exclusively for that thread.
This reduces contention among threads for the heap space.

Fast Object Allocation

Allocating memory for new objects from a TLAB is faster because it avoids synchronization overhead.
Since the TLAB is thread-local, there is no need for locks or coordination with other threads during allocation.

Reduced Fragmentation

By using TLABs, the JVM can allocate objects in contiguous blocks of memory, reducing heap fragmentation and improving cache performance.

Improved Garbage Collection

With TLABs, the JVM can more easily determine which objects were allocated by which threads, aiding in more efficient garbage collection, especially for generational garbage collectors.

Types of Objects in Thread Local Allocation Buffer

The Thread Local Allocation Buffer (TLAB) is used to allocate small objects, typically those that fit entirely within the TLAB's size.

This allocation strategy is primarily aimed at optimizing memory allocation for objects that are frequently created and destroyed, such as short-lived objects.

Here are the main types of objects typically contained in a TLAB:

Small Objects:

- Short-Lived Objects

  These are objects that are created and quickly become unreachable, such as temporary objects used within method executions.

  Examples include local variables, temporary data structures, and intermediate computation results.

- Object Instances

  Small instances of classes, like small data objects, which are frequently allocated and deallocated.

Primitive Arrays:

- Small Arrays

  Arrays of primitive data types (like int[], char[], byte[]) that are small enough to fit within the TLAB size limit.

Small Collections:

- Instances of Collection Classes

  Small instances of collection classes like ArrayList, HashMap, etc., which are often used temporarily within methods and are short-lived.

Java Virtual Machine Stack (thread private)

Structure

The JVM Stack is a runtime data area that stores method invocation information.

Creation time

Each Java Virtual Machine thread has a private *Java Virtual Machine stack*, created at the same time as the thread.

Stack Frame

A Java Virtual Machine stack stores frames (§2.6). A Java Virtual Machine stack is analogous to the stack of a conventional language such as C:

Each method call triggers the creation of a new frame on the stack to store the method's local variables and the return address. Those frames can be stored in the Heap.

Because the Java Virtual Machine stack is never manipulated directly except to push and pop frames, frames may be heap allocated.

The memory for a Java Virtual M achine stack does not need to be contiguous.

Size

This specification permits Java Virtual Machine stacks either to be of a fixed size or to dynamically expand and contract as required by the computation.

If the Java Virtual Machine stacks are of a fixed size, the size of each Java Virtual Machine stack may be chosen independently when that stack is created.

A Java Virtual *Machine implementation may provide the programmer or the user control over the initial size of Java*

Virtual Machine stacks, as well as,
in the case of dynamically expanding or contracting Java Virtual Machine stacks, control over the maximum and minimum sizes.

Exception

- **If the computation in a thread requires a larger Java Virtual Machine stack than is permitted, the Java Virtual Machine throws a StackOverflowError.**
- **If Java Virtual Machine stacks can be dynamically expanded, and expansion is attempted but insufficient memory can be made available to effect the expansion,**
  or if insufficient memory can be made available to create the initial Java Virtual Machine stack for a new thread, the Java Virtual Machine throws an OutOfMemoryError.

Frames

Storage

A *frame* is used to store data and partial results, as well as to perform dynamic linking, return values for methods, and dispatch exceptions.

When is a Stack Frame created?

A new frame is created each time a method is invoked.
A frame is destroyed when its method invocation completes, whether that completion is normal or abrupt (it throws an uncaught exception).
Frames are allocated from the Java Virtual Machine stack (§2.5.2) of the thread creating the frame.

What's contained in a Stack Frame?

Each frame has its own array of local variables (§2.6.1), its own operand stack (§2.6.2), and a reference to the run-time constant pool (§2.5.5) of the class of the current method.

A frame *may be extended with additional implementation-specific* information*, such as debugging information.*

What determines the size of a Stack Frame?

The sizes of the local variable *array* and the operand stack are determined at compile-time and are supplied along with the code for the method associated with the frame (§4.7.3).
Thus the size of the frame data structure depends only on the implementation of the Java Virtual Machine,
and the memory for these structures can be allocated simultaneously on method invocation.

How does the Java Stack operate the Stack Frame?

Only one frame, the frame for the executing method, is active at any point in a given thread of control.
This frame is referred to as the *current frame*, and its method is known as the *current method.* The class in which the current method is defined is the *current class*.
Operations on local variables and the operand stack are typically with reference to the current frame.

A frame ceases to be current if its method invokes another method or if its method completes.
When a method is invoked, a new frame is created and becomes current when control transfers to the new method.
On method return, the current frame passes back the result of its method invocation, if any, to the previous frame.
The current frame is then discarded as the previous frame becomes the current one.
Note that a frame created by a thread is local to that thread and cannot be referenced by any other thread.

Local Variables

The length of Local Variables in Stack Frame?

Each frame (§2.6) contains an array of variables known as its *local variables*.
The length of the local variable array of a frame is determined at compile-time and supplied in the binary representation of a class or interface along with the code for the method associated with the frame (§4.7.3).
so the length of the local variable array does not change during program execution.

Value types of Local Variables

A single local variable can hold a value of type boolean, byte, char, short, int, float, reference, or returnAddress (return to the place where the method was called before).

A pair of local variables can hold a value of type long or double.

Addressing

Local variables are addressed by indexing. The index of the first local variable is zero.

An integer is considered to be an index into the local variable array if and only if that integer is between zero and one less than the size of the local variable array.

Value representation for long and double type

A value of type long or type double occupies two consecutive local variables. Such a value may only be addressed using the lesser index.

For example, a value of type double stored in the local variable array at index *n* actually occupies the local variables with indices *n* and *n*+1;

however, the local variable at index *n*+1 cannot be loaded from.

It can be stored into. However, doing so invalidates the contents of local variable *n*.

The Java Virtual Machine does not require *n* to be even. In intuitive terms, values of types long and double need not be 64-bit aligned in the local variables array.

Implementors are free to decide the appropriate way to represent such values using the two local variables reserved for the value.

Local Variable 0 on method invocation

The Java Virtual Machine uses local variables to pass parameters on method invocation.

On class method invocation, any parameters are passed in consecutive local variables starting from local variable *0*.

On instance method invocation, local variable *0* is always used to pass a reference to the object on which the instance method is being invoked (this in the Java programming language).

Any parameters are subsequently passed in consecutive local variables starting from local variable *1*.

Operand Stacks

1. What's Operand Stacks?

Each frame (§2.6) contains a last-in-first-out (LIFO) stack known as its *operand stack*.

The maximum depth of the operand stack of a frame is determined at compile-time and is supplied along with the code for the method associated with the frame (§4.7.3).

Where it is clear by context, we will sometimes refer to the operand stack of the current frame as simply the operand stack.

All calculation processes in the program are completed with the Operand Stack.

2. What's contained in Operand Stacks?

The operand stack is empty when the frame that contains it is created.

The Java Virtual Machine supplies instructions to load constants or values from local variables or fields onto the operand stack.

Other Java Virtual Machine instructions take operands from the operand stack, operate on them, and push the result back onto the operand stack.

The operand stack is also used to prepare parameters to be passed to methods and to receive method results.

For example, the *iadd* instruction (§*iadd*) adds two int values together. It requires that the int values to be added be the top two values of the operand stack, pushed there by previous instructions.

Both of the int values are popped from the operand stack.

They are added, and their sum is pushed back onto the operand stack.

Subcomputations may be nested on the operand stack, resulting in values that can be used by the encompassing computation.

3. What are the restriction on the Operand Stack?

Each entry on the operand stack can hold a value of any Java Virtual Machine type, including a value of type long or type double.
Values from the operand stack must be operated upon in ways appropriate to their types.

It is not possible, for example, to push two int values and subsequently treat them as a long or to push two float values and subsequently add them with an *iadd* instruction.
A small number of Java Virtual Machine instructions (the *dup* instructions (§*dup*) and *swap* (§*swap*)) operate on run-time data areas as raw values without regard to their specific types;
these instructions are defined in such a way that they cannot be used to modify or break up individual values. These restrictions on operand stack manipulation are enforced through class file verification (§4.10).

4. What's the depth of the Operand Stack?
   At any point in time, an operand stack has an associated depth, where a value of type long or double contributes two units to the depth and a value of any other type contributes one unit.

Native Method Stacks (thread private)

1. What is Native Method Stacks?
   The native method stack is a runtime data area used by the JVM to execute native methods,

   Native methods are methods written in other programming languages, such as C or C++,
   The NMS is separate from the Java Stack which is used to execute Java methods.
   The separation is nessary to prevent native methods from interfering with Java code.

   The Native Method Stack is very similar to the JVM Stack but is only dedicated to native methods.
2. What is the workflow for calling native methods?
   When a thread calls a native method, the thread switches from the Java stack to the NMS, The JVM pushes a frame onto the NMS.
   (The frame contains the arguments to the method, as well as the local variables and return address)
   The JVM then executes the native method. (and if a native method calls back a Java method, the thread leaves the NMS and enters the Java stack again.)

   When the method returns, the JVM pops the frame off the NMS.
3. Is Native Method Stacks necessary?
   Java Virtual Machine implementations that cannot load native methods and that do not themselves rely on conventional stacks need not supply native method stacks.
   If supplied, native method stacks are typically allocated per thread when each thread is created.

1. Is the size of Native Method Stacks fixed?
   This specification permits native method stacks either to be of a fixed size or to dynamically expand and contract as required by the computation.
   If the native method stacks are of a fixed size, the size of each native method stack may be chosen independently when that stack is created.
   A Java Virtual Machine implementation may provide the programmer or the user control over the initial size of the native method stacks,
   as well as, in the case of varying-size native method stacks, control over the maximum and minimum method stack sizes.
2. Associated Exception

a. If the computation in a thread requires a larger native method stack than is permitted, the Java Virtual Machine throws a StackOverflowError.
b. If native method stacks can be dynamically expanded and native method stack expansion is attempted but insufficient memory can be made available,
    or if insufficient memory can be made available to create the initial native method stack for a new thread, the Java Virtual Machine throws an OutOfMemoryError.

Functional Components

The ClassFile Structure

Single ClassFile structure

```
ClassFile {
    u4              magic;
    u2              minor_version;
    u2              major_version;
    u2              constant_pool_count;
    cp_info         constant_pool[constant_pool_count-1];
    u2              access_flags;
    u2              this_class;
    u2              super_class;
    u2              interfaces_count;
    u2              interfaces[interfaces_count];
    u2              fields_count;
    field_info      fields[fields_count];
    u2              methods_count;
    method_info     methods[methods_count];
    u2              attributes_count;
    attribute_info  attributes[attributes_count];
}
```

Byte Stream

A class file consists of a stream of 8-bit bytes. 16-bit and 32-bit quantities are constructed by reading in two and four consecutive 8-bit bytes, respectively.

Multibyte data items are always stored in big-endian order, where the high bytes come first.

This chapter defines the data types u1, u2, and u4 to represent an unsigned one-, two-, or four-byte quantity, respectively. (The u2 type in a Java class file consist of an unsigned 16-bit interger in big-endian byte order)

Fields

1) magic
    Identifies the file as a Java class file. The value is always 0xCAFEBABE.
2) minor_version, major_version
    The values of the minor_version and major_version items are the minor and major version numbers of this class file.
    Together, a major and a minor version number determine the version of the class file format.
    If a class file has major version number M and minor version number m, we denote the version of its class file format as M.m.
3) constant_pool_count
    The value of the constant_pool_count item is equal to the number of entries in the constant_pool table plus one.
    A constant_pool index is considered valid if it is greater than zero and less than constant_pool_count, with the exception for constants of type long and double noted in §4.4.5.

4) constant_pool[]

   The constant_pool is a table of structures (§4.4) representing various string constants, class and interface names, field names,
   and other constants that are referred to within the ClassFile structure and its substructures.
   The format of each constant_pool table entry is indicated by its first "tag" byte.
   The constant_pool table is indexed from 1 to constant_pool_count - 1.

5) access_flags

   The value of the access_flags item is a mask of flags used to denote access permissions to and properties of this class or interface. The interpretation of each flag, when set, is specified in Table 4.1-B.

6) this_class

   The value of the this_class item must be a valid index into the constant_pool table.
   The constant_pool entry at that index must be a CONSTANT_Class_info structure (§4.4.1) representing the class or interface defined by this class file.

7) super_class

   For a class, the value of the super_class item either must be zero or must be a valid index into the constant_pool table.
   If the value of the super_class item is nonzero, the constant_pool entry at that index must be
   a CONSTANT_Class_info structure representing the direct superclass of the class defined by this class file.
   Neither the direct superclass nor any of its superclasses may have the ACC_FINAL flag set in
   the access_flags item of its ClassFile structure.
   If the value of the super_class item is zero, then this class file must represent the class Object, the only class or interface without a direct superclass.
   For an interface, the value of the super_class item must always be a valid index into the constant_pool table.
   The constant_pool entry at that index must be a CONSTANT_Class_info structure representing the
   class Object.

8) interfaces_count

   The value of the interfaces_count item gives the number of direct superinterfaces of this class or interface type.

9) interfaces[]

   Each value in the interfaces array must be a valid index into the constant_pool table.
   The constant_pool entry at each value of interfaces[i], where $0 \le i <$ interfaces_count,
   must be a CONSTANT_Class_info structure representing an interface that is a direct superinterface of this class or interface type,
   in the left-to-right order given in the source for the type.

10) fields_count

    The value of the fields_count item gives the number of field_info structures in the fields table.
    The field_info structures represent all fields, both class variables and instance variables, declared by this class or interface type.

11) fields[]

    Each value in the fields table must be a field_info structure (§4.5) giving a complete description of a field in this class or interface. The fields table includes only those fields that are declared by this class or interface. It does not include items representing fields that are inherited from superclasses or superinterfaces.

12) methods_count

    The value of the methods_count item gives the number of method_info structures in the methods table.

13) methods[]

    Each value in the methods table must be a method_info structure (§4.6) giving a complete description of a method in this class or interface. If neither of the ACC_NATIVE and ACC_ABSTRACT flags are set in
    the access_flags item of a method_info structure, the Java Virtual Machine instructions implementing the

method are also supplied.

The method_info structures represent all methods declared by this class or interface type, including instance methods, class methods, instance initialization methods (§2.9.1), and any class or interface initialization method (§2.9.2). The methods table does not include items representing methods that are inherited from superclasses or superinterfaces.

14) attributes_count

The value of the attributes_count item gives the number of attributes in the attributes table of this class.

15) attributes[]

Each value of the attributes table must be an attribute_info structure (§4.7).

The attributes defined by this specification as appearing in the attributes table of a ClassFile structure are listed in Table 4.7-C.

The rules concerning attributes defined to appear in the attributes table of a ClassFile structure are given in §4.7.

The rules concerning non-predefined attributes in the attributes table of a ClassFile structure are given in §4.7.1.

The Constant Pool

Purpose

The Java constant pool is a collection of constants that are used by JVM to run the code of a class.
It's a runtime data structure that is part of the class file format.

Java Virtual Machine instructions do not rely on the run-time layout of classes, interfaces, class instances, or arrays. Instead, instructions refer to symbolic information in the constant_pool table.

Entries

All constant_pool table entries have the following general format:

```
cp_info {
    u1 tag;
    u1 info[];
}
```

Each entry in the constant_pool table must begin with a 1-byte tag indicating the kind of constant denoted by the entry.
Each tag byte must be followed by two or more bytes giving information about the specific constant. The format of the additional information depends on the tag byte,
that is, the content of the info array varies with the value of tag.

tag Types

Constant pool tags

In a class file whose version number is *v*, each entry in the constant_pool table must have a tag that was first defined in version *v* or earlier of the class file format (§4.1).

That is, each entry must denote a kind of constant that is approved for use in the class file.
Table 4.4-B lists each tag with the first version of the class file format in which it was defined.
Also shown is the version of the Java SE Platform which introduced that version of the class file format.

Constant pool tags (by tag)

Constant Kind	Tag	`class` file format	Java SE
`CONSTANT_Utf8`	1	45.3	1.0.2
`CONSTANT_Integer`	3	45.3	1.0.2
`CONSTANT_Float`	4	45.3	1.0.2

Constant Kind	Tag	class file format	Java SE
CONSTANT_Long	5	45.3	1.0.2
CONSTANT_Double	6	45.3	1.0.2
CONSTANT_Class	7	45.3	1.0.2
CONSTANT_String	8	45.3	1.0.2
CONSTANT_Fieldref	9	45.3	1.0.2
CONSTANT_Methodref	10	45.3	1.0.2
CONSTANT_InterfaceMethodref	11	45.3	1.0.2
CONSTANT_NameAndType	12	45.3	1.0.2
CONSTANT_MethodHandle	15	51.0	7
CONSTANT_MethodType	16	51.0	7
CONSTANT_Dynamic	17	55.0	11
CONSTANT_InvokeDynamic	18	51.0	7
CONSTANT_Module	19	53.0	9
CONSTANT_Package	20	53.0	9

Loadable constant pool tags

Some entries in the constant_pool table are *loadable* because they represent entities that can be pushed onto the stack at run time to enable further computation.

In a class file whose version number is *v*, an entry in the constant_pool table is loadable if it has a tag that was first deemed to be loadable in version *v* or earlier of the class file format.
Table 4.4-C lists each tag with the first version of the class file format in which it was deemed to be loadable.
Also shown is the version of the Java SE Platform which introduced that version of the class file format.

In every case *except CONSTANT_Class, a tag was first deemed to be loadable in the same version of the class file format that first defined the tag.*

Loadable constant pool tags

Constant Kind	Tag	class file format	Java SE
CONSTANT_Integer	3	45.3	1.0.2
CONSTANT_Float	4	45.3	1.0.2
CONSTANT_Long	5	45.3	1.0.2
CONSTANT_Double	6	45.3	1.0.2
CONSTANT_Class	7	49.0	5.0
CONSTANT_String	8	45.3	1.0.2
CONSTANT_MethodHandle	15	51.0	7
CONSTANT_MethodType	16	51.0	7
CONSTANT_Dynamic	17	55.0	11

info Types

- The CONSTANT_Class_info structure is used to represent a class or an interface:

```
CONSTANT_Class_info {
    u1 tag;
```

u2 name_index;
}
The items of the CONSTANT_Class_info structure are as follows:
tag
The tag item has the value CONSTANT_Class (7).
name_index
The value of the name_index item must be a valid index into the constant_pool table.
The constant_pool entry at that index must be a CONSTANT_Utf8_info structure (§4.4.7) representing a valid binary class or interface name encoded in internal form (§4.2.1).

Because arrays are objects, the opcodes *anewarray* and *multianewarray* - but not the opcode *new* - can reference array "classes" via CONSTANT_Class_info structures in the constant_pool table.

For such array classes, the name of the class is the descriptor of the array type (§4.3.2).

For example, the class name representing the two-dimensional array type int[][] is [[I, while the class name representing the type Thread[] is [Ljava/lang/Thread;.

An array type descriptor is valid only if it represents 255 or fewer dimensions.

- The CONSTANT_String_info structure is used to represent constant objects of the type String:

CONSTANT_String_info {
u1 tag;
u2 string_index;
}
The items of the CONSTANT_String_info structure are as follows:
tag
The tag item has the value CONSTANT_String (8).
string_index
The value of the string_index item must be a valid index into the constant_pool table.
The constant_pool entry at that index must be a CONSTANT_Utf8_info structure (§4.4.7) representing the sequence of Unicode code points to which the String object is to be initialized.

DirectBuffer

Definition

A direct buffer refers to a buffer's underlying data allocated on a memory area where OS functions can directly access it.
A non-direct buffer refers to a buffer whose underlying data is a byte array that is allocated in the Java heap area.

it reduces the amount of work to be done during I/O since a native buffer is ready as-is to be passed to the kernel, while using non-native buffers requires an additional pass.

Usage

https://wiki.sei.cmu.edu/confluence/display/java/OBJ53-J.+Do+not+use+direct+buffers+for+short-lived%2C+infrequently+used+objects
This compliant solution uses an indirect buffer to allocate the short-lived, infrequently used object.
The heavily used buffer appropriately continues to use a nonheap, non-garbage-collected direct buffer.

Do not use direct buffers for short-lived, infrequently used objects

```
ByteBuffer rarelyUsedBuffer = ByteBuffer.allocate(8192);
// Use rarelyUsedBuffer once
ByteBuffer heavilyUsedBuffer = ByteBuffer.allocateDirect(8192);
// Use heavilyUsedBuffer many times
```

The Internal Form of Names

Binary Class and Interface Names

Class and interface names that appear in class file structures are always represented in a fully qualified form known as *binary names* (JLS §13.1).

Such names are always represented as CONSTANT_Utf8_info structures (§4.4.7) and thus may be drawn, where not further constrained,
from the entire Unicode codespace. Class and interface names are referenced from
those CONSTANT_NameAndType_info structures (§4.4.6) which have such names as part of their descriptor (§4.3), and from all CONSTANT_Class_info structures (§4.4.1).

For historical reasons, the syntax of binary names that appear in class file structures differs from the syntax of binary names documented in JLS §13.1.
In this internal form, the ASCII periods (.) that normally separate the identifiers which make up the binary name are replaced by ASCII forward slashes (/).
The identifiers themselves must be unqualified names (§4.2.2).

For example, the normal binary name of class Thread is java.lang.Thread. In the internal form used in descriptors in the class file format,
a reference to the name of class Thread is implemented using a CONSTANT_Utf8_info structure representing the string java/lang/Thread.

Descriptors and Signatures

A *descriptor* is a string representing the type of a field or method.
Descriptors are represented in the class file format using modified UTF-8 strings (§4.4.7) and thus may be drawn, where not further constrained, from the entire Unicode codespace.
A *signature* is a string representing the generic type of a field or method, or generic type information for a class declaration.

Field Descriptors

Structure

A *field descriptor* represents the type of a class, instance, or local variable. It is a series of characters generated by the grammar:

```
FieldDescriptor:
    FieldType

FieldType:
    BaseType
    ObjectType
    ArrayType

BaseType:
    B
    C
    D
    F
    I
    J
    S
    Z

ObjectType:
    L ClassName ;

ArrayType:
    [ ComponentType
```

```
ComponentType:
    FieldType
```

The characters of *BaseType*, the L and ; of *ObjectType*, and the [of *ArrayType* are all ASCII characters.

The *ClassName* represents a binary class or interface name encoded in internal form (§4.2.1).

The interpretation of field descriptors as types is as shown in Table 4.2.

A field descriptor representing an array type is valid only if it represents a type with 255 or fewer dimensions.

Interpretation of *FieldType* characters

***BaseType* Character**	**Type**	**Interpretation**
B	byte	signed byte
C	char	Unicode character code point in the Basic Multilingual Plane, encoded with UTF-16
D	double	double-precision floating-point value
F	float	single-precision floating-point value
I	int	integer
J	long	long integer
L *ClassName* ;	reference	an instance of class *ClassName*
S	short	signed short
Z	boolean	true or false
[	reference	one array dimension

The field descriptor of an instance variable of type int is simply I.

The field descriptor of an instance variable of type Object is Ljava/lang/Object;. Note that the internal form of the binary name for class Object is used.

The field descriptor of an instance variable that is a multidimensional double array, double d[][][], is [[[D.

Method Descriptors

Structure

A *method descriptor* represents the parameters that the method takes and the value that it returns:

```
MethodDescriptor:
    ( ParameterDescriptor* ) ReturnDescriptor
```

A *parameter descriptor* represents a parameter passed to a method:

```
ParameterDescriptor:
    FieldType
```

A *return descriptor* represents the type of the value returned from a method. It is a series of characters generated by the grammar:

```
ReturnDescriptor:
    FieldType
    VoidDescriptor

VoidDescriptor:
    V
```

The character V indicates that the method returns no value (its return type is void).

A method descriptor is valid only if it represents method parameters with a total length of 255 or less, where that length includes the contribution for this in the case of instance or interface method invocations.

The total length is calculated by summing the contributions of the individual parameters,
where a parameter of type long or double contributes two units to the length and a parameter of any other type contributes one unit.

The method descriptor for the method:
```
Object m(int i, double d, Thread t) {..}
```
is (IDLjava/lang/Thread;)Ljava/lang/Object;. Note that the internal forms of the binary names of Thread and Object are used.

The method descriptor for m is the same whether m is a class method or an instance method.
Although an instance method is passed this, a reference to the current class instance, in addition to its intended parameters,
that fact is not reflected in the method descriptor.
The reference to this is passed implicitly by the method invocation instructions of the Java Virtual Machine that invoke instance methods (§2.6.1).
A reference to this is not passed to a class method.

Garbage Collection

Concept

Purpose

Garbage Collection is the process of reclaiming the runtime unused memory by destroying the unused objects.
Java Garbage Collection is the process by which Java programs perform automatic memory management.

You can say that at any point in time, the heap memory consists of two types of objects:
- Live - these objects are being used and referenced from somewhere else
- Dead - these objects are no longer used or referenced from anywhere

The garbage collector finds these unused objects and deletes them to free up memory.

Dereference an object

The main objective of Garbage Collection is to free heap memory by destroying the objects that don't contain a reference.
When there are no references to an object, it is assumed to be dead and no longer needed. So the memory occupied by the object can be reclaimed.

There are various ways in which the references to an object can be released to make it a candidate for Garbage Collection.
Some of them are:
- By making a reference null
```
    Student student = new Student();
    student = null;
```
- By assigning a reference to another
```
    Student studentOne = new Student();
    Student studentTwo = new Student();
    studentOne = studentTwo; // now the first object referred by studentOne is available for garbage
    collection
```
- By using an anonymous object
```
    register(new Student());
```

Process

Java garbage collection is an automatic process. The programmer does not need to explicitly mark objects to be deleted.

The garbage collection implementation lives in the JVM. Each JVM can implement its own version of garbage collection.

However, it should meet the standard JVM specification of working with the objects present in the heap memory, marking or identifying the unreachable objects, and destroying them with compaction.

Garbage Collection Roots

Garbage collectors work on the concept of *Garbage Collection Roots* (GC Roots) to identify live and dead objects. Examples of such Garbage Collection roots are:

- Classes loaded by system class loader (not custom class loaders)
- Live threads
- Local variables and parameters of the currently executing methods
- Local variables and parameters of JNI methods

- Global JNI reference

  A reference to a Java object that persists beyond the scope of a single JNI function call.

```
#include <jni.h>

// Create a global reference
jobject createGlobalReference(JNIEnv *env, jobject localRef) {
    // Create and return a global reference
    return (*env)->NewGlobalRef(env, localRef);
}

// Use the global reference
void useGlobalReference(JNIEnv *env, jobject globalRef) {
    // Perform operations with the global reference
}

// Delete the global reference
void deleteGlobalReference(JNIEnv *env, jobject globalRef) {
    (*env)->DeleteGlobalRef(env, globalRef);
}
```

- Objects used as a monitor for synchronization
- Objects held from garbage collection by JVM for its purposes

The garbage collector traverses the whole object graph in memory, starting from those Garbage Collection Roots and following references from the roots to other objects.

How to Identify Garbage

Reference Counting

Reference Counting is a simple but slow scheme.
In this scheme, each object contains a reference counter.
Every time a reference is attached to that object, its reference counter is increased. Every time a reference goes out or is set to null, its counter is decreased.
Once the reference counter of an object becomes zero, that storage will be released.

But one shortcoming is that if objects circularly refer to each other they can have nonzero reference counts. Below is an example:

```
public class ReferenceCountingGC {
    public Object instance;
    public static void main(String...strings) {
        ReferenceCountingGC a = new ReferenceCountingGC(); // 1st instance
        ReferenceCountingGC b = new ReferenceCountingGC(); // 2nd instance
        a.instance = b;
        b.instance = a;
        a = null;
        b = null;
    }
    }
```

It creates two instances. The first instance has two references, one is from a and the other is from b.instance.

The second instance also has two references. After two variables are set to null, the two instances still have one reference from their members.

In this case, their reference counter will never be zero.

Another drawback is that reference counting scheme requires extra space for each object to store its reference counter and extra process resource to deal with counter.

Reachability Analysis

The basic idea of Reachability Analysis is to trace which objects are reachable by a chain of references from "root" objects.

If two objects reference each other but are not on the root reference chain, they will also be considered unreachable.

The "root" is called GC Roots and the path from root to the certain object is called reference chain.

If one object cannot be reachable from any GC Roots, it will be considered as garbage and the storage will be released.

1)	Unavailable and unreachable:	Recycling objects
2)	Unavailable and reachable：	There may be a memory leak in this situation (the reference was cleared during object definition, but the object still has a reference elsewhere)
3)	available and reachable：	normal use

Finalization and Termination Condition

finalizer

The finalize method provided by the root Object class. Simply put, this is called before the garbage collection for a particular object.

The main purpose of a finalizer is to release resources used by objects before they're removed from the memory.

If the finalize() method were completely empty, the JVM would treat the object as if it didn't have a finalizer.

Termination

In reality, the time at which the garbage collector calls finalizers is dependent on the JVM's implementation and the system's conditions, which are out of our control.

To make garbage collection happen on the spot, we'll take advantage of the System.gc method.

In real-world systems, we should never invoke that explicitly, for a number of reasons:

It's costly

It doesn't trigger the garbage collection immediately – it's just a hint for the JVM to start GC

JVM knows better when GC needs to be called

If we need to force GC, we can use jconsole for that.

drawbacks

Despite the benefits they bring in, finalizers come with many drawbacks.

Let's have a look at several problems we'll be facing when using finalizers to perform critical actions.

1) The first noticeable issue is the lack of promptness.

We cannot know when a finalizer runs since garbage collection may occur anytime.

By itself, this isn't a problem because the finalizer still executes, sooner or later.

However, system resources aren't unlimited. Thus, we may run out of resources before a clean-up happens, which may result in a system crash.

2) The last problem we'll be talking about is the lack of exception handling during finalization.
   If a finalizer throws an exception, the finalization process stops, leaving the object in a corrupted state without any notification.

3) finalizers are very expensive.
   When creating an object, also called a referent, that has a finalizer, the JVM creates an accompanying reference object of type java.lang.ref.Finalizer.
   JVM must perform many more operations when constructing and destroying objects containing a non-empty finalizer.

- After the referent is ready for garbage collection, the JVM marks the reference object Finalizer as ready for processing and puts it into a reference queue.
  We can access this queue via the static field queue in the java.lang.ref.Finalizer class.
- Meanwhile, a special daemon thread called Finalizer keeps running and looks for objects in the reference queue.
  When it finds one, it removes the reference object Finalizer from the queue and calls the finalizer on the referent.
- Perform Reachability Analysis
  During the next garbage collection cycle, the referent will be discarded – when it's no longer referenced from a reference object Finalizer.

If a thread keeps producing objects at a high speed, which is what happened in our example, the Finalizer thread cannot keep up.
Eventually, the memory won't be able to store all the objects, and we end up with an OutOfMemoryError.

Example:
```
public class CrashedFinalizable {
    public static void main(String[] args) throws ReflectiveOperationException {
        for (int i = 0; ; i++) {
            new CrashedFinalizable();
             if ((i % 1_000_000) == 0) {
                 Class<?> finalizerClass = Class.forName("java.lang.ref.Finalizer");
                 Field queueStaticField = finalizerClass.getDeclaredField("queue");
                 queueStaticField.setAccessible(true);
                 ReferenceQueue<Object> referenceQueue = (ReferenceQueue)
             queueStaticField.get(null);

                 Field queueLengthField =
             ReferenceQueue.class.getDeclaredField("queueLength");
                 queueLengthField.setAccessible(true);
                 long queueLength = (long) queueLengthField.get(referenceQueue);
                 System.out.format("There are %d references in the queue%n", queueLength);
             }
        }
    }

    @Override
    protected void finalize() {
        System.out.print("");
    }
}
```
Results:
```
...
There are 21914844 references in the queue
There are 22858923 references in the queue
There are 24202629 references in the queue
There are 24621725 references in the queue
There are 25410983 references in the queue
There are 26231621 references in the queue
There are 26975913 references in the queue
Exception in thread "main" java.lang.OutOfMemoryError: GC overhead limit exceeded
    at java.lang.ref.Finalizer.register(Finalizer.java:91)
    at java.lang.Object.<init>(Object.java:37)
    at com.baeldung.finalize.CrashedFinalizable.<init>(CrashedFinalizable.java:6)
    at com.baeldung.finalize.CrashedFinalizable.main(CrashedFinalizable.java:9)

Process finished with exit code 1
```

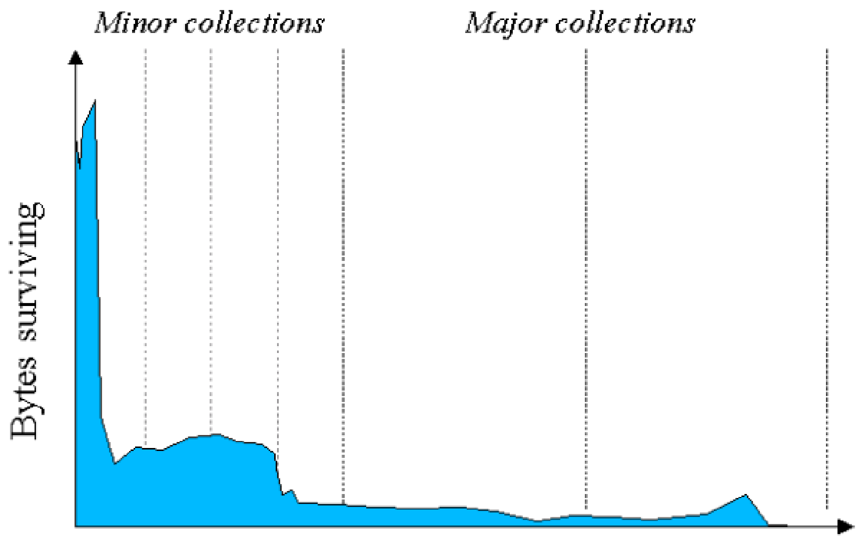
Generational Garbage Collection

Performance

Java Garbage Collectors implement a *generational garbage collection strategy* that categorizes objects by age.

Having to mark and compact all the objects in a JVM is inefficient.
As more and more objects are allocated, the list of objects grows, leading to longer garbage collection times.

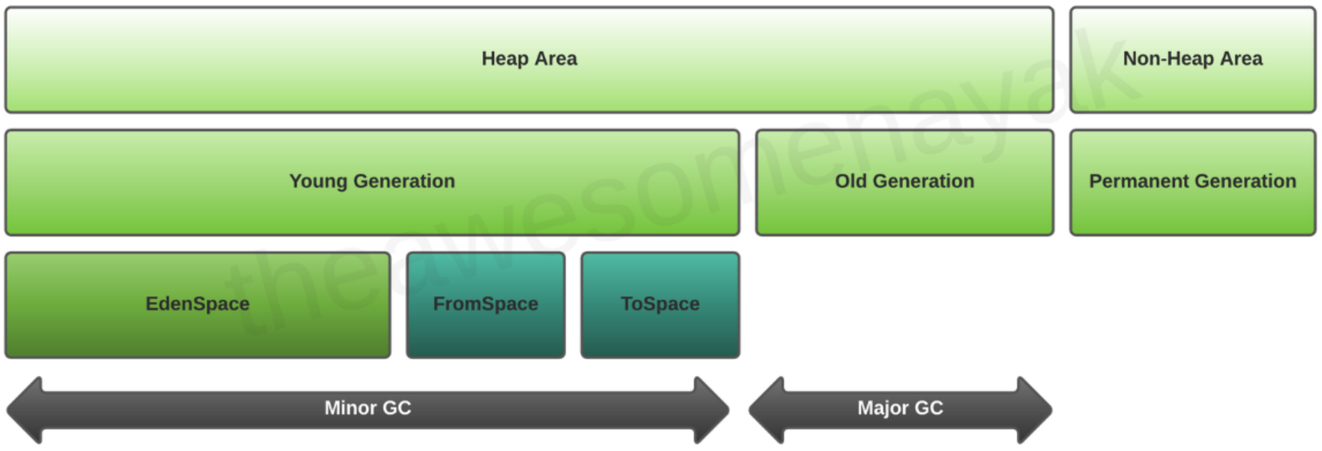
Empirical analysis of applications has shown that most objects in Java are short lived.



In the above example, the Y axis shows the number of bytes allocated and the X axis shows the number of bytes allocated over time.
As you can see, fewer and fewer objects remain allocated over time.
In fact most objects have a very short life as shown by the higher values on the left side of the graph.
This is why Java categorizes objects into generations and performs garbage collection accordingly.

Process

The heap memory area in the JVM is divided into three sections:



1) Young Generation

Newly created objects start in the Young Generation. The Young Generation is further subdivided into:

- Eden space — all new objects start here, and initial memory is allocated to them
- Survivor Spaces **(**FromSpace and ToSpace**)** — objects are moved here from Eden after surviving one garbage collection cycle.

Trigger Timing

When Eden space is filled with objects, a Minor GC is performed.

Garbage Collection

When objects are garbage collected from the Young Generation, it is a *minor garbage collection event*.
All the dead objects are deleted, and all the live objects are moved to one of the Survivor Spaces.
Minor GC also checks the objects in a Survivor Space, and moves them to the other Survivor Space.

Process

- Eden has all objects (live and dead)
- Minor GC occurs - all dead objects are removed from Eden. All live objects are moved to S1 (FromSpace). Eden and S2 are now empty.
- New objects are created and added to Eden. Some objects in Eden and S1 become dead.
- Minor GC occurs - all dead objects are removed from Eden and S1. All live objects are moved to S2 (ToSpace). Eden and S1 are now empty.

So, at any time, one of the survivor spaces is always empty.
When the surviving objects reach a certain threshold of moving around the survivor spaces, they are moved to the Old Generation.
You can use the -Xmn flag to set the size of the Young Generation.

2) Old Generation

Objects that are long-lived are eventually moved from the Young Generation to the Old Generation.
This is also known as Tenured Generation, and contains objects that have remained in the Survivor Spaces for a long time.
There is a threshold defined for the tenure of an object which decides how many garbage collection cycles it can survive before it is moved to the Old Generation.
Since Java uses generational garbage collection, the more garbage collection events an object survives, the further it gets promoted in the heap.
It starts in the young generation and eventually ends up in the tenured generation if it survives long enough.
When objects are garbage collected from the Old Generation, it is a major garbage collection event.

Trigger Timing

Major GC is triggered when the Old Generation is full or nearly full, or when a full GC is explicitly requested

via JVM options or by the application.
It can also be triggered by certain events, like system memory pressure.

You can use the -Xms and -Xmx flags to set the size of the initial and maximum size of the Heap memory.

3) Permanent Generation
   Metadata such as classes and methods are stored in the Permanent Generation.
   It is populated by the JVM at runtime based on classes in use by the application.
   Classes that are no longer in use may be garbage collected from the Permanent Generation.

   You can use the -XX:PermGen and -XX:MaxPermGen flags to set the initial and maximum size of the Permanent Generation.

4) MetaSpace
   Starting with Java 8, the MetaSpace memory space replaces the PermGen space.
   The implementation differs from the PermGen and this space of the heap is now automatically resized.

   This avoids the problem of applications running out of memory due to the limited size of the PermGen space of the heap.
   The Metaspace memory can be garbage collected and the classes that are no longer used can be automatically cleaned when the Metaspace reaches its maximum size.

Mark and Sweep

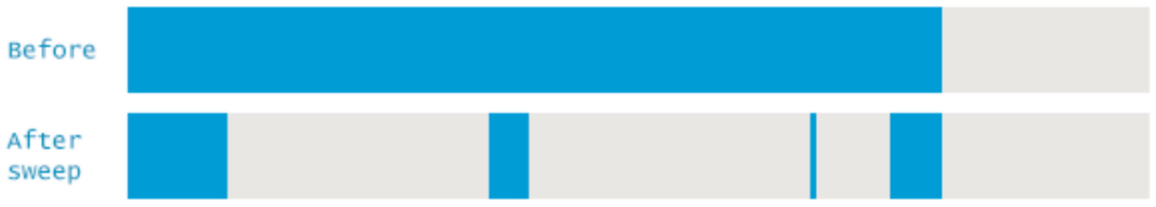
1. What's mark and sweep algorithm?
   It is initial and very basic algorithm which runs in two stages:

Marking live objects	find out all objects that are still alive.
Removing unreachable objects	get rid of everything else – the supposedly dead and unused objects.

   To start with, GC defines some specific objects as Garbage Collection Roots.
   Now GC traverses the whole object graph in your memory, starting from those roots and following references from the roots to other objects.
   Every object the GC visits is marked as alive.

   The application threads need to be stopped for the marking to happen as it cannot really traverse the graph if it keeps changing.
   It is called Stop The World pause.

2. Second stage
   Second stage is for getting rid of unused objects to freeup memory. This can be done in variety of ways e.g.
   1) Normal deletion
      Normal deletion removes unreferenced objects to free space and leave referenced objects and pointers.
      The memory allocator (kind of hashtable) holds references to blocks of free space where new object can be allocated.
      It is often referred as mark-sweep algorithm.

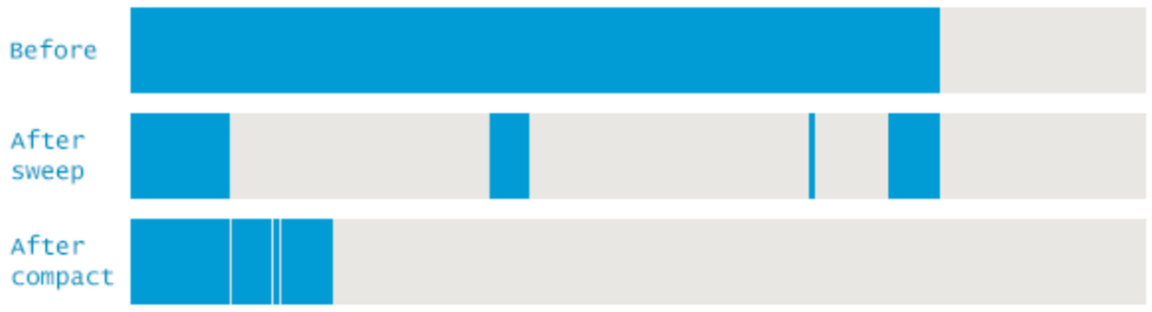


   2) Deletion with compacting

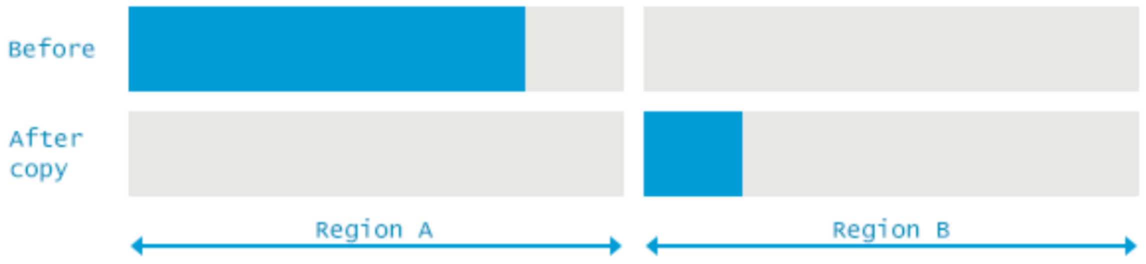
Only removing unused objects is not efficient because blocks of free memory is scattered across storage area and cause OutOfMemoryError,
if created object big enough and does not find large enough memory block.

To solve this issue, after deleting unreferenced objects, compacting is done on the remaining referenced objects.
Here compacting refers the process of moving referenced object together. This makes new memory allocation much easier and faster.

It is often referred as mark-sweep-compact algorithm.



3) Deletion with copying
It is very similar to mark and compacing approach as they relocate all live objects as well.
The important difference is that the target of relocation is a different memory region.
It is often reffred as mark-copy algorithm.



Choosing a garbage collector

Performance factors

1) Throughput
The percentage of total time spent in useful application activity versus memory allocation and garbage collection.
For example, if your throughput is 95%, that means the application code is running 95% of the time and garbage collection is running 5% of the time.
You want higher throughput for any high-load business application.
2) Latency
Application responsiveness, which is affected by garbage collection pauses.
In any application interacting with a human or some active process (such as a valve in a factory), you want the lowest possible latency.
3) Footprint
The working set of a process, measured in pages and cache lines.

Choose a garbage collector

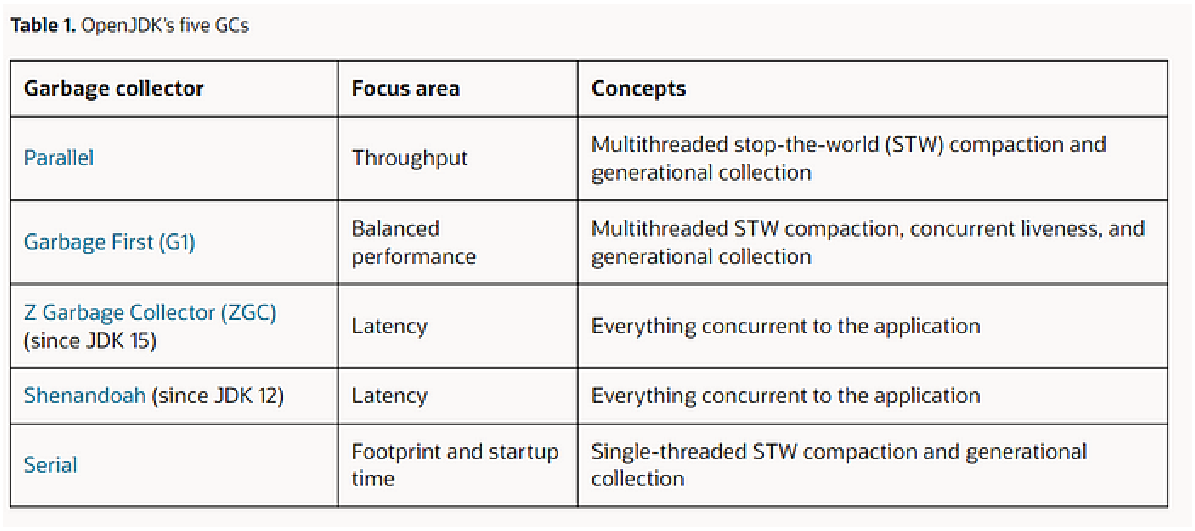


Table 1. OpenJDK's five GCs

Garbage collector	Focus area	Concepts
Parallel	Throughput	Multithreaded stop-the-world (STW) compaction and generational collection
Garbage First (G1)	Balanced performance	Multithreaded STW compaction, concurrent liveness, and generational collection
Z Garbage Collector (ZGC) (since JDK 15)	Latency	Everything concurrent to the application
Shenandoah (since JDK 12)	Latency	Everything concurrent to the application
Serial	Footprint and startup time	Single-threaded STW compaction and generational collection

Garbage Collector

Reference:

Java Garbage Collection Algorithms [till Java 9]

https://howtodoinjava.com/java/garbage-collection/all-garbage-collection-algorithms/

Garbage Collection in Java – What is GC and How it Works in the JVM

https://www.freecodecamp.org/news/garbage-collection-in-java-what-is-gc-and-how-it-works-in-the-jvm

Serial GC(Young Generation)

Definition

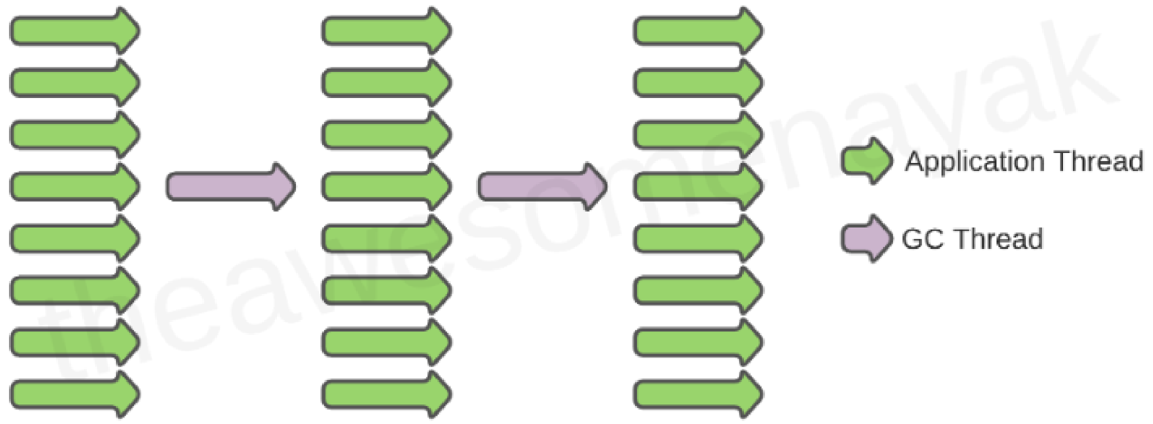
Uses a single thread for garbage collection, suitable for small applications with low memory.

Versions

Introduced in JDK 1.2

Stop-the-World

The application threads are stopped (stop-the-world pause) during the young generation garbage collection.



Algorithm

Young Generation Collection:

- **Algorithm**: Mark-Copy (Copying)
- **Phases**:
  - **Mark**: Identifies live objects in the Eden space and from-space of the Survivor spaces.
  - **Copy**: Copies live objects to the to-space of Survivor spaces.
    The memory occupied by dead objects is essentially ignored and remains occupied until the next young generation collection.

The original space containing dead objects (usually Eden) is then considered free space for future allocations. This effectively discards the dead objects and reclaims their memory.

Old Generation Collection:

- **Algorithm**: Mark-Sweep-Compact
- **Phases**:
  - **Initial Mark**: A quick, stop-the-world phase that marks objects directly reachable from the roots.
  - **Mark**: A single-threaded phase that continues to mark all live objects reachable from the roots.
  - **Sweep**: A single-threaded phase that reclaims space by sweeping away dead objects.
  - **Compact**: A single-threaded phase that compacts the memory by moving live objects together to eliminate fragmentation.

Jvm options

-XX:+UseSerialGC	Use Serial Garbage Collector

Serial Old GC (Old Generation)

Definition

Serial Old GC refers to the garbage collection strategy used for the old (tenured) generation of the heap.

Versions

Introduced in JDK 1.2

Algorithm

Old Generation Collection:

- **Algorithm**: Mark-Sweep-Compact
- **Phases**:
  - **Initial Mark**: A quick, stop-the-world phase that marks objects directly reachable from the roots.
  - **Mark**: A single-threaded phase that continues to mark all live objects reachable from the roots.
  - **Sweep**: A single-threaded phase that reclaims space by sweeping away dead objects.
  - **Compact**: A single-threaded phase that compacts the memory by moving live objects together to eliminate fragmentation.

Parallel GC

Definition

Versions:

Introduced in	JDK 1.4
Enhanced in	JDK 5 and JDK 6 with additional parallelism and adaptiveness
Default	JDK 1.4 to JDK 8

Stop-the-World

Although it uses multiple threads, the application threads are stopped (stop-the-world pause) during the young generation garbage collection.

Algorithm

Young Generation Collection:

- **Algorithm**: Parallel Mark-Copy (Copying)
- **Phases**:
  - **Mark**: Uses multiple threads to identify live objects in the Eden space and from-space of the Survivor spaces.
  - **Copy**: Uses multiple threads to copy live objects to the to-space of Survivor spaces.

Old Generation Collection:

- **Algorithm**: Parallel Mark-Sweep-Compact

- **Phases**:
    - **Initial Mark**: A quick, stop-the-world phase that marks objects directly reachable from the roots using multiple threads.
    - **Mark**: Uses multiple threads to continue marking all live objects reachable from the roots.
    - **Sweep**: Uses multiple threads to reclaim space by sweeping away dead objects.
    - **Compact**: Uses multiple threads to compact the memory by moving live objects together to eliminate fragmentation.

x

Jvm options

-XX:+UseParallelGC Use Parallel Garbage Collector

-XX:+UseParallelOldGC Use Parallel Old Garbage Collector，It is same as Parallel GC except that it uses multiple threads for both Young Generation and Old Generation.

Parallel Old GC (Old Generation)

Definition

Parallel Old GC refers to the garbage collection strategy used for the old (tenured) generation of the heap.
The old generation is where long-lived objects are eventually promoted after surviving multiple garbage collection cycles in the young generation. The key characteristics of Parallel Old GC include:

Versions:

Introduced in JDK 1.4

Algorithm

Old Generation Collection:

- **Algorithm**: Parallel Mark-Sweep-Compact
- **Phases**:
    - **Initial Mark**: A quick, stop-the-world phase that marks objects directly reachable from the roots using multiple threads.
    - **Mark**: Uses multiple threads to continue marking all live objects reachable from the roots.
    - **Sweep**: Uses multiple threads to reclaim space by sweeping away dead objects.
    - **Compact**: Uses multiple threads to compact the memory by moving live objects together to eliminate fragmentation.

CMS (Concurrent Mark Sweep) GC

Definition

Aimed at applications requiring low pause times, it performs most of its work concurrently with the application threads. No compaction is performed in CMS GC.

Versions:

Introduced in	JDK 1.4
Deprecated in	JDK 9
Removed in	JDK 14

Algorithm

Young Generation Collection:

- **Algorithm**: ParNew GC (Parallel Copying)
- **Phases**:
    - **Mark**: Identifies live objects in the Eden space and from-space of the Survivor spaces using multiple threads.
    - **Copy**: Copies live objects to the to-space of Survivor spaces using multiple threads.

Old Generation Collection:

- **Algorithm**: Mark-Sweep
- **Phases**:
  - **Initial Mark**: A quick, stop-the-world phase that marks objects directly reachable from the roots.
  - **Concurrent Mark**: A concurrent phase that continues to mark live objects reachable from those marked in the initial phase.
  - **Remark**: A short stop-the-world phase that finalizes marking of any objects that were modified during the concurrent mark phase.
  - **Concurrent Sweep**: A concurrent phase that reclaims space by sweeping away dead objects.
  - **Concurrent Reset**: Prepares the CMS collector for the next cycle by resetting data structures.

Jvm options

-XX:+UseConcMarkSweepGC	use CMS GC
-XX:+UseCMSInitiating\OccupancyOnly	Indicates that you want to solely use occupancy as a criterion for starting a CMS collection operation.
-XX:CMSInitiating\OccupancyFraction=70	Sets the percentage CMS generation occupancy to start a CMS collection cycle.
-XX:CMSTriggerRatio=70	This is the percentage of MinHeapFreeRatio in CMS generation that is allocated prior to a CMS cycle starts.
-XX:CMSTriggerPermRatio=90	Sets the percentage of MinHeapFreeRatio in the CMS permanent generation that is allocated before starting a CMS collection cycle.
-XX:CMSWaitDuration=2000	Use the parameter to specify how long the CMS is allowed to wait for young collection.
-XX:+UseParNewGC	Elects to use the parallel algorithm for young space collection.
-XX:+CMSConcurrentMTEnabled	Enables the use of multiple threads for concurrent phases.
-XX:ConcGCThreads=2	Sets the number of parallel threads used for the concurrent phases.
-XX:ParallelGCThreads=2	Sets the number of parallel threads you want used for stop-the-world phases.
-XX:+CMSIncrementalMode	Enable the incremental CMS (iCMS) mode.
-XX:+CMSClassUnloadingEnabled	If this is not enabled, CMS will not clean permanent space.
-XX:+ExplicitGCInvokes\Concurrent	This allows System.gc() to trigger concurrent collection instead of a full garbage collection cycle.

Garbage First GC

Definition

Designed for large heap applications with a focus on low pause times, it divides the heap into regions and prioritizes areas with the most garbage.

Versions:

Introduced in	JDK 7 (experimental)
Made default in	JDK 9

The G1 collector is a parallel, concurrent, and incrementally compacting low-pause garbage collector.

This approach involves segmenting the memory heap into multiple small regions (typically 2048).
Each region is marked as either young generation (further devided into eden regions or survivor regions) or old generation.
This allows the GC to avoid collecting the entire heap at once, and instead approach the problem incrementally. It means

that only a subset of the regions is considered at a time.

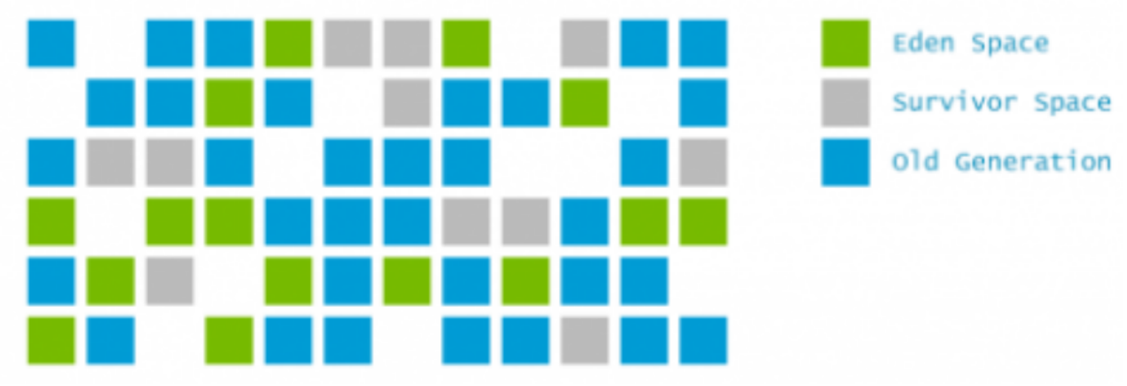
Algorithm

Young Generation Collection:

- **Algorithm**: Parallel Mark-Copy (Copying)
- **Phases**:
    - **Mark**: Uses multiple threads to identify live objects.
    - **Copy**: Uses multiple threads to copy live objects to new regions.

Old Generation Collection:

- **Algorithm**: Region-Based Mark-Sweep-Compact
- **Phases**:
    - **Initial Mark**: A quick, stop-the-world phase that marks objects directly reachable from the roots.
    - **Root Region Scanning**: A concurrent phase that scans the root regions to identify additional live objects.
    - **Concurrent Mark**: A concurrent phase that continues to mark live objects reachable from those marked in the initial phase.
    - **Remark**: A short stop-the-world phase that finalizes marking of any objects that were modified during the concurrent mark phase.
    - **Cleanup**: A stop-the-world phase that identifies and processes empty regions.
    - **Concurrent Cleanup**: A concurrent phase that reclaims space from dead objects and prepares for the next GC cycle.
    - **Copying (Evacuation)**: Live objects are copied (evacuated) from regions being collected to empty regions, compacting the heap.



Jvm options

-XX:+UseG1GC	Use the G1 Garbage Collector
-XX:G1HeapRegionSize=16m	Size of the heap region. The value will be a power of two and can range from 1MB to 32MB. The goal is to have around 2048 regions based on the minimum Java heap size.
-XX:MaxGCPauseMillis=200	Sets a target value for desired maximum pause time. The default value is 200 milliseconds. The specified value does not adapt to your heap size.
-XX:G1ReservePercent=5	This determines the minimum reserve in the heap.
-XX:G1ConfidencePercent=75	This is the confidence coefficient pause prediction heuristics.
-XX:GCPauseIntervalMillis=200	This is the pause interval time slice per MMU in milliseconds.

Epsilon Garbage Collector

Definition

Epsilon is a do-nothing (no-op) garbage collector.
It handles memory allocation but does not implement any actual memory reclamation mechanism. Once the available Java heap is exhausted, the JVM shuts down.
Version:
Introduced in JDK 11
It can be used for ultra-latency-sensitive applications, where developers know the application memory footprint exactly, or even have (almost) completely garbage-free applications.
Usage of the Epsilon GC in any other scenario is otherwise discouraged.
Jvm options

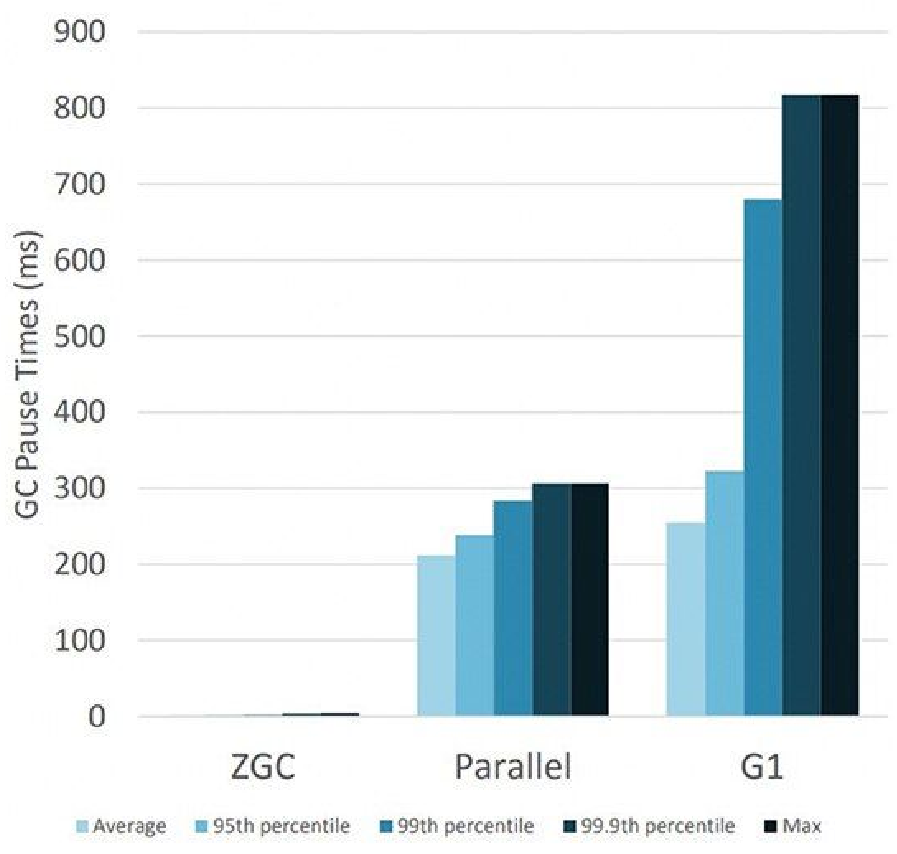
-XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC	Use the Epsilon Garbage Collector

ZGC

Definition
Versions:

Introduced in	JDK 11 (experimental)
Made production-ready in	JDK 15

It is intended for applications which require low latency (less than 10 ms pauses) and/or use a very large heap (multi-terabytes).
The primary goals of ZGC are low latency, scalability, and ease of use.
To achieve this, ZGC allows a Java application to continue running while it performs all garbage collection operations.
By default, ZGC uncommits unused memory and returns it to the operating system.
Thus, ZGC brings a significant improvement over other traditional GCs by providing extremely low pause times (typically within 2ms).



Algorithm

Young Generation Collection:

- **Algorithm**: Concurrent Mark-Relocate
- **Phases**:
  - **Concurrent Mark**: A concurrent phase that marks live objects throughout the heap.
  - **Concurrent Relocate**: A concurrent phase that relocates live objects to new memory locations.

Old Generation Collection:

- **Algorithm**: Concurrent Mark-Relocate
- **Phases**:
  - **Concurrent Mark**: A concurrent phase that marks live objects throughout the heap.
  - **Concurrent Relocate**: A concurrent phase that relocates live objects to new memory locations.
  - **Concurrent Remap**: A concurrent phase that updates references to relocated objects.
  - **Concurrent Cleanup**: A concurrent phase that reclaims space from dead objects.

Jvm options

-XX:+UnlockExperimentalVMOptions -XX:+UseZGC	Use the Epsilon Garbage Collector

Shenandoah

Definition

Shenandoah's key advantage over G1 is that it does more of its garbage collection cycle work concurrently with the application threads.

Versions:

Introduced in	JDK 12 (experimental)
Made production-ready in	JDK 15

Algorithm

Young Generation Collection:

- **Algorithm**: Concurrent Mark-Copy (Copying)
- **Phases**:
  - **Initial Mark:** This is a quick, stop-the-world phase that identifies objects directly reachable from the roots.
  - **Concurrent Mark:** Shenandoah continues to mark live objects concurrently with the application threads.
  - **Concurrent Evacuation:** Shenandoah concurrently moves live objects to new regions within the Young Generation to compact memory and reduce fragmentation.

    When Shenandoah moves an object from one memory location to another (evacuation), it updates any references to that object immediately.

    This integrated approach ensures that there is no need for a separate reference updating phase.

Old Generation Collection:

- **Algorithm**: Concurrent Mark-Compact
- **Phases**:
  - **Initial Mark:** Similar to the Young Generation, this is a quick, stop-the-world phase that marks objects directly reachable from the roots.
  - **Concurrent Mark:** Shenandoah continues marking live objects concurrently with the application threads.
  - **Final Mark:** This is a short stop-the-world phase to ensure all live objects are

marked.

- **Concurrent Cleanup:** Shenandoah performs cleanup of unreachable objects concurrently.
- **Concurrent Evacuation:** Live objects are moved to new regions within the Old Generation concurrently to compact the heap.
- **Concurrent Update References:** Shenandoah updates references to the newly evacuated objects concurrently.

In the Old Generation, objects have more complex reference patterns and typically more references pointing to them. Therefore, a separate "Concurrent Update References" phase is necessary to handle the more extensive reference updating required after objects have been moved during compaction. This helps ensure that all references throughout the heap are correctly updated without introducing significant pauses.

Jvm options

-XX:+UnlockExperimentalVMOptions -XX:+UseShenandoahGC   Use the Shenandoah Garbage Collector

Java IO/NIO

Reference:
https://www.linkedin.com/pulse/java-sockets-io-blocking-non-blocking-asynchronous-aliaksandr-liakh

The POSIX definitions (Portable Operating System Interface)

Systems that implement POSIX

Unix, Linux, Mac OS X, BSD, Solaris, AIX, etc.

Sockets

Sockets are endpoints to perform two-way communication by TCP and UDP protocols.
Java sockets APIs are adapters for the corresponding functionality of the operating systems.
Sockets communication in POSIX-compliant operating systems (Unix, Linux, Mac OS X, BSD, Solaris, AIX, etc.) is performed by Berkeley sockets.
Sockets communication in Windows is performed by Winsock that is also based on Berkeley sockets with additional functionality to comply with the Windows programming model.

Zero Copy

Zero-copy is a computer operation that minimizes the number of data copies made between memory areas.
It is a technique used in various systems, including networking and file I/O, to improve performance and reduce CPU usage.
In traditional data transfer methods, data is typically copied multiple times between different buffers in the operating system and application memory space.
Zero-copy aims to eliminate these redundant copies.

Zero-copy techniques vary depending on the specific system or operation, but the core idea is to avoid unnecessary data copying by allowing different parts of the system to access the same memory buffer directly.
Here are some common scenarios where zero-copy is used:

Network Communication

In network communication, zero-copy is often used to efficiently send and receive data over a network without multiple copies. Here's how it works:

Traditional Method:

Data is read from the disk into a user-space buffer.
Data is copied from the user-space buffer to a kernel-space buffer.
Data is sent from the kernel-space buffer to the network interface.

Zero-Copy Method:

Data is read directly from the disk into a kernel-space buffer.

Data is sent directly from the kernel-space buffer to the network interface.

This method reduces the number of data copies from two to one, significantly improving performance.

File I/O

In file I/O operations, zero-copy can be used to transfer data between files or between a file and a network socket without copying data multiple times between user space and kernel space.

Traditional Method:

Data is read from the disk into a user-space buffer.

Data is written from the user-space buffer back to the disk or sent over a network.

Zero-Copy Method:

Data is read from the disk into a kernel-space buffer.

Data is directly transferred from the kernel-space buffer to the disk or network socket.

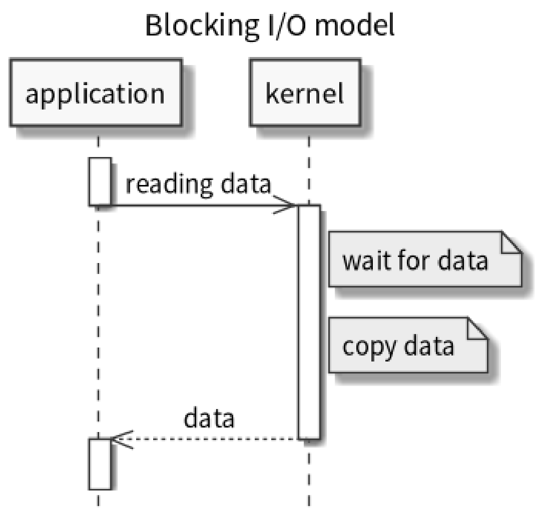
DMA (Direct Memory Access)

Zero-copy often involves the use of DMA, a feature in modern hardware that allows peripherals to access system memory independently of the CPU.

This means that data can be transferred directly between memory and a device (such as a network card or disk) without involving the CPU, further reducing overhead and improving performance.

Common IO models for the POSIX-compliant operating systems

blocking I/O model (BIO)



blocking system call

In the blocking I/O model, the application makes a blocking system call until data is received at the kernel and is copied from kernel space into user space (user thread).

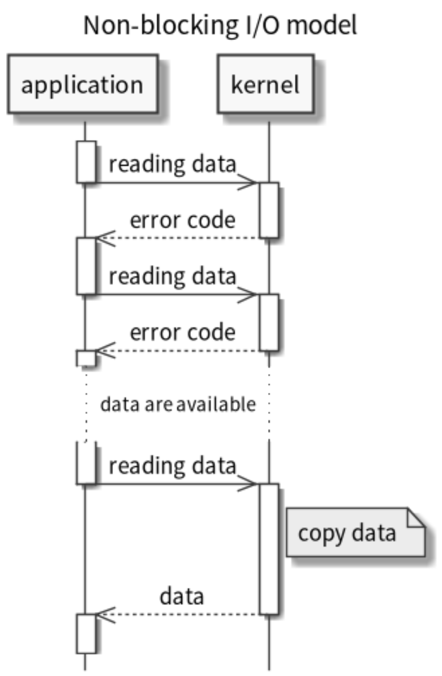
Pros

- The simplest I/O model to implement

Cons

- The application is blocked

non-blocking I/O model (NIO)



system call

In the non-blocking I/O model the application makes a system call that immediately returns one of two responses:

- if the I/O operation can be completed immediately, the data is returned
- if the I/O operation can't be completed immediately, an error code is returned indicating that the I/O operation would block or the device is temporarily unavailable

To complete the I/O operation, the application should busy-wait (make repeating system calls) until completion.

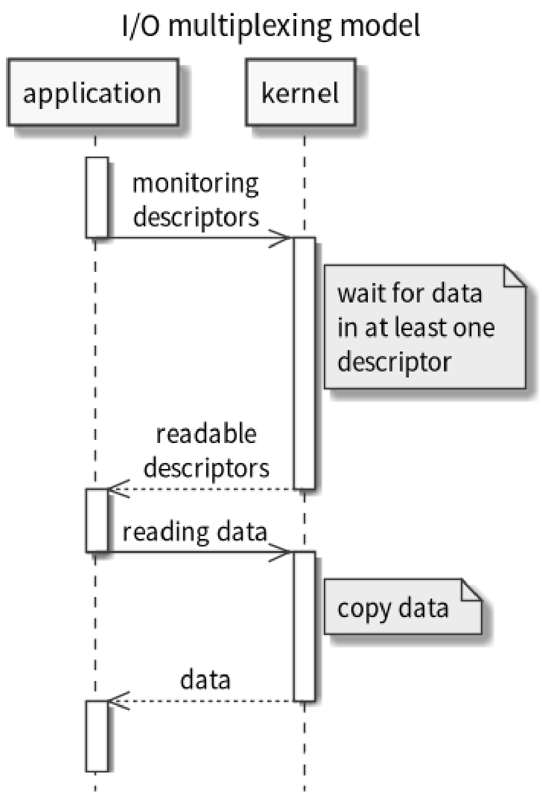
Pros:

- The application isn't blocked

Cons:

- The application should busy-wait until completion, that would cause many user-kernel context switches.
- This model can introduce I/O latency because there can be a gap between the data availability in the kernel and the data reading by the application.

I/O multiplexing model



blocking select system call

In the I/O multiplexing model (also known as the non-blocking I/O model with blocking notifications),
the application makes a blocking select system call (blocking notifications) to start monitoring activity on many descriptors.

File descriptor in IO operations:

A file descriptor is a unique identifier or reference that the operating system assigns to a file when it is opened.
It allows programs to interact with files, sockets, or other input/output (I/O) resources.
The file descriptor is used by the operating system to keep track of the file and perform operations on it.

For each descriptor, it's possible to request notification of its readiness for certain I/O operations (connection, reading or writing, error occurrence, etc.).

non-blocking call

When the select system call returns, indicating that at least one descriptor is ready,
the application makes a non-blocking call and copies the data from kernel space into user space.

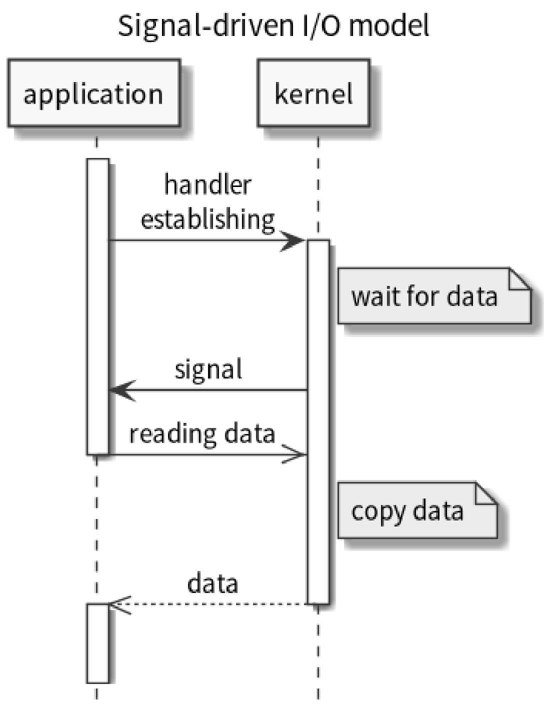
Pros:

- It's possible to perform I/O operations on multiple descriptors in one thread

Cons:

- The application is still blocked on the select system call
- Not all operating systems support this model efficiently

signal-driven I/O model



non-blocking call

In the signal-driven I/O model the application makes a non-blocking call and registers a signal handler.

The application does not constantly check (or "poll") the status of the file descriptor, which would be busy-waiting.

Instead, it registers a signal handler with the operating system for a specific signal (like SIGIO in UNIX).

The signal handler is used to handle I/O events efficiently by responding to signals sent by the operating system, allowing the application to perform necessary I/O actions without constantly checking for readiness.

When a file descriptor is ready for an I/O operation, a signal is generated for the application.

non-blocking copying

Then the signal handler copies the data from kernel space into user space.

The data copying process within the signal handler is non-blocking if the file descriptor is set to non-blocking mode.

The signal handler is responsible for performing I/O operations without causing the thread to wait or block.

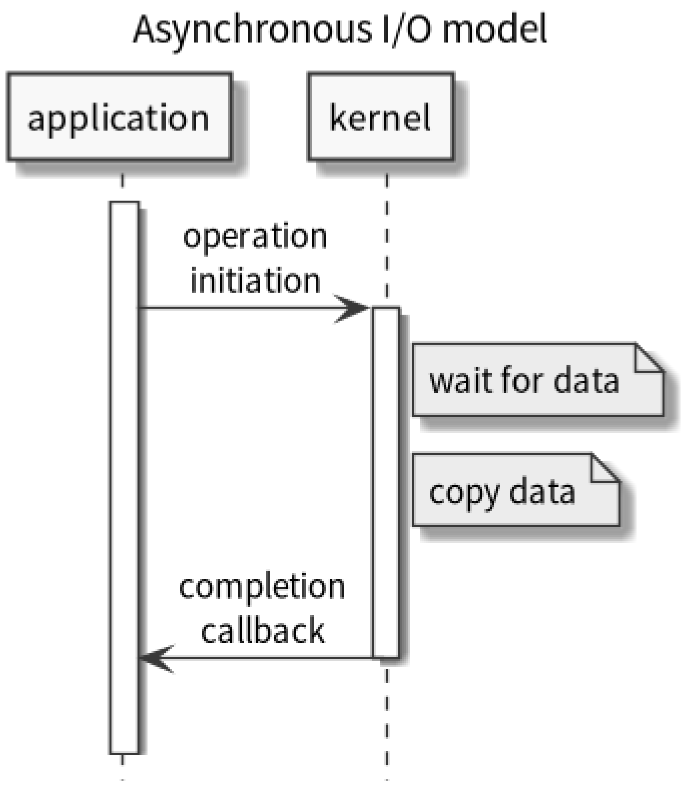
Pros:

- The application isn't blocked
- Signals can provide good performance

Cons:

- Not all operating systems support signals

asynchronous I/O model (AIO)



non-blocking call

In the asynchronous I/O model (also known as the overlapped I/O model) the application makes the non-blocking call and starts a background operation in the kernel.

callback

When the operation is completed (data are received at the kernel and are copied from kernel space into user space), a completion callback is executed to finish the I/O operation.

A difference between the asynchronous I/O model and the signal-driven I/O model is that with signal-driven I/O, the kernel tells the application when an I/O operation can be initiated,
but with the asynchronous I/O model, the kernel tells the application when an I/O operation is completed.

Pros:

- The application isn't blocked
- This model can provide the best performance

Cons:

- The most complicated I/O model to implement
- Not all operating systems support this model efficiently

Java IO API

Java IO API is based on streams (InputStream, OutputStream) that represent blocking, one-directional data flow.

Java NIO API

Java NIO stands for New Input/Output. It is a Java API that provides features for intensive I/O operations.
NIO was introduced in J2SE 1.4, and it is designed to provide access to the low-level I/O operations of modern operating systems.

Java NIO consists of several subpackages, each addressing specific functionalities:

java.nio	Provides core classes like Buffer and related utility classes.
java.nio.channels	Contains classes for working with various channels (file channels, network sockets etc.)

java.nio.charset	Deals with character set encoding and decoding.

Components

Java NIO API is based on the Channel, Buffer, Selector classes, that are adapters to low-level I/O operations of operating systems.

- Channel

  The Channel class represents a connection to an entity (hardware device, file, socket, software component, etc) that is capable of performing I/O operations (reading or writing).

  In comparison with uni-directional streams, channels are bi-directional.
- Buffer

  The Buffer class is a fixed-size data container with additional methods to read and write data.

  All Channel data are handled through Buffer but never directly:

  all data that are sent to a Channel are written into a Buffer,

  all data that are received from a Channel are read into a Buffer.

  In comparison with streams, that are byte-oriented, channels are block-oriented. Byte-oriented I/O is simpler but for some I/O entities can be rather slow.

  Block-oriented I/O can be much faster but is more complicated.
- Selector

  The Selector class allows subscribing to events from many registered SelectableChannel objects in a single call.

  When events arrive, a Selector object dispatches them to the corresponding event handlers.

FileChannel Usage

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class FileChannelExample {
    public static void main(String[] args) {
        Path path = Paths.get("example.txt");

        try (FileChannel fileChannel = FileChannel.open(path, StandardOpenOption.READ)) {
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            int bytesRead = fileChannel.read(buffer);

            while (bytesRead != -1) {
                buffer.flip(); // Switch to read mode
                while (buffer.hasRemaining()) {
                    System.out.print((char) buffer.get());
                }
                buffer.clear(); // Prepare buffer for the next read
                bytesRead = fileChannel.read(buffer);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

DatagramChannel Usage

```
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.DatagramChannel;

public class DatagramChannelExample {
    public static void main(String[] args) {
        try {
            DatagramChannel datagramChannel = DatagramChannel.open();
            datagramChannel.bind(new InetSocketAddress(9999));

            ByteBuffer buffer = ByteBuffer.allocate(1024);

            while (true) {
                buffer.clear(); // Prepare buffer for the next read
                datagramChannel.receive(buffer);
                buffer.flip(); // Switch to read mode

                while (buffer.hasRemaining()) {
                    System.out.print((char) buffer.get());
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Java NIO2 API

Java NIO2 is a new API introduced in Java SE 7 for file and directory operations. It provides a number of improvements over the older Java I/O API.

Components

Java NIO2 API is based on asynchronous channels (AsynchronousServerSocketChannel, AsynchronousSocketChannel, etc) that support asynchronous I/O operations
(connecting, reading or writing, errors handling).

The asynchronous channels provide two mechanisms to control asynchronous I/O operations.

- The first mechanism is by returning a java.util.concurrent.Future object, which models a pending operation and can be used to query the state and obtain the result.
- The second mechanism is by passing to the operation a java.nio.channels.CompletionHandler object,
  which defines handler methods that are executed after the operation has completed or failed. The provided API for both mechanisms are equivalent.

Asynchronous channels provide a standard way of performing asynchronous operations platform-independently.
However, the amount that Java sockets API can exploit native asynchronous capabilities of an operating system, will depend on the support for that platform.

Socket echo server

Blocking IO echo server

In the following example, the blocking I/O model is implemented in an echo server with Java IO API.

The ServerSocket.accept method blocks until a connection is accepted.
The InputStream.read method blocks until input data are available, or a client is disconnected.
The OutputStream.write method blocks until all output data are written.

```
public class IoEchoServer {

   public static void main(String[] args) throws IOException {
       ServerSocket serverSocket = new ServerSocket(7000);

       while (active) {
           Socket socket = serverSocket.accept(); // blocking

           InputStream is = socket.getInputStream();
           OutputStream os = socket.getOutputStream();

           int read;
```

```
            byte[] bytes = new byte[1024];
            while ((read = is.read(bytes)) != -1) { // blocking
                os.write(bytes, 0, read); // blocking
            }

            socket.close();
        }

        serverSocket.close();
    }
}
```

Blocking NIO echo server

In the following example, the blocking I/O model is implemented in an echo server with Java NIO API.

The ServerSocketChannel and SocketChannel objects are configured in the blocking mode by default.
The ServerSocketChannel.accept method blocks and returns a SocketChannel object when a connection is accepted.
The ServerSocket.read method blocks until input data are available, or a client is disconnected.
The ServerSocket.write method blocks until all output data are written.

```
public class NioBlockingEchoServer {

   public static void main(String[] args) throws IOException {
       ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
       serverSocketChannel.bind(new InetSocketAddress("localhost", 7000));

       while (active) {
           SocketChannel socketChannel = serverSocketChannel.accept(); // blocking

           ByteBuffer buffer = ByteBuffer.allocate(1024);
           while (true) {
               buffer.clear();
               int read = socketChannel.read(buffer); // blocking
               if (read < 0) {
                   break;
               }

               buffer.flip();
               socketChannel.write(buffer); // blocking
           }

           socketChannel.close();
       }

       serverSocketChannel.close();
   }
}
```

Non-blocking NIO echo server

In the following example, the non-blocking I/O model is implemented in an echo server with Java NIO API.

The ServerSocketChannel and SocketChannel objects are explicitly configured in the non-blocking mode.
The ServerSocketChannel.accept method doesn't block and returns null if no connection is accepted yet or a SocketChannel object otherwise.
The ServerSocket.read doesn't block and returns 0 if no data are available or a positive number of bytes read otherwise.
The ServerSocket.write method doesn't block if there is free space in the socket's output buffer.

```
public class NioNonBlockingEchoServer {

   public static void main(String[] args) throws IOException {
       ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
       serverSocketChannel.configureBlocking(false);
       serverSocketChannel.bind(new InetSocketAddress(7000));
```

```
        while (active) {
            SocketChannel socketChannel = serverSocketChannel.accept(); // non-blocking
            if (socketChannel != null) {
                socketChannel.configureBlocking(false);

                ByteBuffer buffer = ByteBuffer.allocate(1024);
                while (true) {
                    buffer.clear();
                    int read = socketChannel.read(buffer); // non-blocking
                    if (read < 0) {
                        break;
                    }

                    buffer.flip();
                    socketChannel.write(buffer); // can be non-blocking
                }

                socketChannel.close();
            }
        }

        serverSocketChannel.close();
    }
}
```

Multiplexing NIO echo server

In the following example, the multiplexing I/O model is implemented in an echo server Java NIO API.

During the initialization, multiple ServerSocketChannel objects, that are configured in the non-blocking mode,
are registered on the same Selector object with the SelectionKey.OP_ACCEPT argument to specify that an event of connection acceptance is interesting.

In the main loop, the Selector.select method blocks until at least one of the registered events occurs.
Then the Selector.selectedKeys method returns a set of the SelectionKey objects for which events have occurred.
Iterating through the SelectionKey objects, it's possible to determine what I/O event (connect, accept, read, write) has happened
and which sockets objects (ServerSocketChannel, SocketChannel) have been associated with that event.

Indication of a selection key that a channel is ready for some operation is a hint, not a guarantee.

```
public class NioMultiplexingEchoServer {

   public static void main(String[] args) throws IOException {
       final int ports = 8;
       ServerSocketChannel[] serverSocketChannels = new ServerSocketChannel[ports];

       Selector selector = Selector.open();

       for (int p = 0; p < ports; p++) {
           ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
           serverSocketChannels[p] = serverSocketChannel;
           serverSocketChannel.configureBlocking(false);
           serverSocketChannel.bind(new InetSocketAddress("localhost", 7000 + p));

           serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);
       }

       while (active) {
           selector.select(); // blocking

           Iterator<SelectionKey> keysIterator = selector.selectedKeys().iterator();
           while (keysIterator.hasNext()) {
               SelectionKey key = keysIterator.next();

               if (key.isAcceptable()) {
                   accept(selector, key);
               }

               if (key.isReadable()) {
                   keysIterator.remove();
                   read(selector, key);
               }
               if (key.isWritable()) {
                   keysIterator.remove();
                   write(key);
               }
           }
       }

       for (ServerSocketChannel serverSocketChannel : serverSocketChannels) {
           serverSocketChannel.close();
       }
   }
}
```

When a SelectionKey object indicates that a connection acceptance event has happened,
it's made the ServerSocketChannel.accept call (which can be a non-blocking) to accept the connection.
After that, a new SocketChannel object is configured in the non-blocking mode and is registered on the same Selector object with the SelectionKey.OP_READ argument
to specify that now an event of reading is interesting.

```
private static void accept(Selector selector, SelectionKey key) throws IOException {
    ServerSocketChannel serverSocketChannel = (ServerSocketChannel) key.channel();
    SocketChannel socketChannel = serverSocketChannel.accept(); // can be non-blocking
    if (socketChannel != null) {
        socketChannel.configureBlocking(false);
        socketChannel.register(selector, SelectionKey.OP_READ);
    }
}
```

When a SelectionKey object indicates that a reading event has happened,
it's made a the SocketChannel.read call (which can be a non-blocking) to read data from the SocketChannel object into a new ByteByffer object.
After that, the SocketChannel object is registered on the same Selector object with the SelectionKey.OP_WRITE argument to specify that now an event of write is interesting.
Additionally, this ByteBuffer object is used during the registration as an attachment.

```
private static void read(Selector selector, SelectionKey key) throws IOException {
    SocketChannel socketChannel = (SocketChannel) key.channel();

    ByteBuffer buffer = ByteBuffer.allocate(1024);
    socketChannel.read(buffer); // can be non-blocking

    buffer.flip();
    socketChannel.register(selector, SelectionKey.OP_WRITE, buffer);
}
```

When a SelectionKeys object indicates that a writing event has happened,
it's made the SocketChannel.write call (which can be a non-blocking) to write data to the SocketChannel object from the ByteByffer object,
extracted from the SelectionKey.attachment method.
After that, the SocketChannel.cloase call closes the connection.

```
    private static void write(SelectionKey key) throws IOException {
        SocketChannel socketChannel = (SocketChannel) key.channel();

        ByteBuffer buffer = (ByteBuffer) key.attachment();

        socketChannel.write(buffer); // can be non-blocking
        socketChannel.close();
    }
```

After every reading or writing the SelectionKey object is removed from the set of the SelectionKey objects to prevent its reuse.
But the SelectionKey object for connection acceptance is not removed to have the ability to make the next similar operation.

Asynchronous NIO2 echo server

In the following example, the asynchronous I/O model is implemented in an echo server with Java NIO2 API.
The AsynchronousServerSocketChannel, AsynchronousSocketChannel classes here are used with the completion handlers mechanism.

The AsynchronousServerSocketChannel.accept method initiates an asynchronous connection acceptance operation.

```
public class Nio2CompletionHandlerEchoServer {

   public static void main(String[] args) throws IOException {
       AsynchronousServerSocketChannel serverSocketChannel = AsynchronousServerSocketChannel.open();
       serverSocketChannel.bind(new InetSocketAddress(7000));

       AcceptCompletionHandler acceptCompletionHandler = new
AcceptCompletionHandler(serverSocketChannel);
       serverSocketChannel.accept(null, acceptCompletionHandler);

       System.in.read();
   }
}
```

When a connection is accepted (or the operation fails), the AcceptCompletionHandler class is called,
which by the AsynchronousSocketChannel.read(ByteBuffer destination, A attachment, CompletionHandler<Integer,? super A> handler) method
initiates an asynchronous read operation from the AsynchronousSocketChannel object to a new ByteBuffer object.

```
class AcceptCompletionHandler implements CompletionHandler<AsynchronousSocketChannel, Void> {

   private final AsynchronousServerSocketChannel serverSocketChannel;

   AcceptCompletionHandler(AsynchronousServerSocketChannel serverSocketChannel) {
       this.serverSocketChannel = serverSocketChannel;
   }

   @Override
   public void completed(AsynchronousSocketChannel socketChannel, Void attachment) {
       serverSocketChannel.accept(null, this); // non-blocking

       ByteBuffer buffer = ByteBuffer.allocate(1024);
       ReadCompletionHandler readCompletionHandler = new ReadCompletionHandler(socketChannel, buffer);
       socketChannel.read(buffer, null, readCompletionHandler); // non-blocking
   }

   @Override
   public void failed(Throwable t, Void attachment) {
       // exception handling
   }
```

```
    }
```

When the read operation completes (or fails), the ReadCompletionHandler class is called,
which by the AsynchronousSocketChannel.write(ByteBuffer source, A attachment, CompletionHandler<Integer,? super A> handler) method
initiates an asynchronous write operation to the AsynchronousSocketChannel object from the ByteBuffer object.

```
    class ReadCompletionHandler implements CompletionHandler<Integer, Void> {

       private final AsynchronousSocketChannel socketChannel;
       private final ByteBuffer buffer;

       ReadCompletionHandler(AsynchronousSocketChannel socketChannel, ByteBuffer buffer) {
           this.socketChannel = socketChannel;
           this.buffer = buffer;
       }

       @Override
       public void completed(Integer bytesRead, Void attachment) {
           WriteCompletionHandler writeCompletionHandler = new WriteCompletionHandler(socketChannel);
           buffer.flip();
           socketChannel.write(buffer, null, writeCompletionHandler); // non-blocking
       }

       @Override
       public void failed(Throwable t, Void attachment) {
           // exception handling
       }
    }
```

When the write operation completes (or fails), the WriteCompletionHandler class is called, which by the AsynchronousSocketChannel.close method closes the connection.

```
    class WriteCompletionHandler implements CompletionHandler<Integer, Void> {

       private final AsynchronousSocketChannel socketChannel;

       WriteCompletionHandler(AsynchronousSocketChannel socketChannel) {
           this.socketChannel = socketChannel;
       }

       @Override
       public void completed(Integer bytesWritten, Void attachment) {
           try {
               socketChannel.close();
           } catch (IOException e) {
               // exception handling
           }
       }

       @Override
       public void failed(Throwable t, Void attachment) {
           // exception handling
       }
    }
```

In this example, asynchronous I/O operations are performed without attachment,
because all the necessary objects (AsynchronousSocketChannel, ByteBuffer) are passed as constructor arguments for the appropriate completion handlers.

Java Locks

Reference:

https://medium.com/@abhirup.acharya009/managing-concurrent-access-optimistic-locking-vs-pessimistic-locking-0f6a64294db7

https://medium.com/nerd-for-tech/optimistic-vs-pessimistic-locking-strategies-b5d7f4925910

Common Lock Classification

Optimistic locking and Pessimistic locking

Optimistic locking

Optimistic locking is a concurrency control mechanism that assumes that multiple transactions will not attempt to modify the same data at the same time.

That is to say, it is believed that there are more read operations and fewer write operations, so the read or write data will not be locked.

However, when updating, it will be determined whether someone else has updated this data duiring this period.

To implement optimistic locking in Java, you can use the jakarta.persistence.Version annotation on a field in your entity class.

Pessimistic locking

Pessimistic locking is a concurrency control mechanism that prevents data conflicts by locking resources before they are accessed or modified.

That is to say, It is believed that there are more write operations and fewer read operations, so both read and write data will be locked.

Pessimistic locks in java

Pessimistic locking can be implemented in Java using the synchronized keyword.

Shared Lock and Exclusive Lock

Shared lock

A Shared Lock allows multiple threads to read a data item simultaneously, but prevents any thread from writing to it.

This is useful when multiple threads need to access the same data item for reading purposes, but only one thread needs to be able to write to it at a time.

Shared locks are also known as read locks.

Shared locks in java

Here are some examples of shared locks in Java:

- ReentrantLock:

  A ReentrantLock can be used to implement a shared lock.

  To do this, you would call the lock() method to acquire the lock and the unlock() method to release the lock.
- ReadWriteLock:

  A ReadWriteLock can be used to implement a shared lock.

  To do this, you would call the readLock() method to acquire a shared lock and the writeLock() method to acquire a write lock.
- StampedLock:

  A StampedLock can be used to implement a shared lock.

  To do this, you would call the tryOptimisticRead() method to acquire a shared lock and the unlockRead() method to release the lock.

Exclusive lock

An exclusive lock allows only one thread to read or write a data item at a time.

This is useful when a thread needs to modify a data item and ensure that no other thread is reading or writing to it at the same time.

Exclusive locks are also known as write locks.

Exclusive locks in java

In Java, there are multiple ways to implement exclusive locks. Here are some of them:

- synchronized keyword:

  The synchronized keyword can be used to lock an object or a class.

  When a thread acquires a lock on an object, no other thread can access that object until the first thread releases the lock.

- ReentrantLock:
  The ReentrantLock class is a more powerful way to implement exclusive locks.
  It allows a thread to acquire a lock multiple times and release it multiple times.
- StampedLock:
  The StampedLock class was introduced in Java 8.
  It provides a more flexible way to control access to a shared resource by allowing multiple threads to read the resource concurrently.
- Semaphores:
  Semaphores are a type of lock that can be used to control access to a shared resource.
  They allow a limited number of threads to access the resource at the same time.

Difference between shared lock and exclusive lock

Feature	Number of threads that can read the data item	Number of threads that can write the data item	Use case
Shared Lock	Multiple	Zero	When multiple threads need to read the same data item, but only one thread needs to be able to write to it at a time.
Exclusive Lock	One	One	When a thread needs to modify a data item and ensure that no other thread is reading or writing to it at the same time.

Fair lock and unfair lock

Fair lock

In Java, a fair lock is a lock that guarantees that threads acquire the lock in the order in which they requested it.
This means that no thread can "jump the queue" and acquire the lock before another thread that has been waiting longer.
Fair locks are typically used in situations where it is important to ensure that all threads have a fair chance of acquiring the lock.
For example, a fair lock might be used to protect a shared resource that is accessed by multiple threads.

Fair locks in java

There are two fair locks in Java:

- ReentrantLock
  This is the most commonly used fair lock in Java. It is implemented using the AbstractQueuedSynchronizer (AQS) class.
  ReentrantLock allows multiple locking operations by the same thread and supports nested locking, which means a thread can lock the same resource multiple times.
- StampedLock
  This lock was introduced in Java 8.
  It provides a more flexible way to control access to a shared resource by allowing multiple threads to read the resource concurrently.
  StampedLock also supports optimistic locking, which allows a thread to acquire a lock without blocking if the resource is not currently locked.

ReentrantLock is a fair lock by default, which means that threads waiting to acquire the lock will be granted the lock in the order in which they requested it.
StampedLock is not a fair lock by default, but it can be configured to be fair by passing the fair parameter to the constructor.

Unfair lock

An unfair lock, on the other hand, does not guarantee fairness.

This means that a thread may be able to acquire the lock before another thread that has been waiting longer.

Unfair locks are typically used in situations where performance is more important than fairness.

For example, an unfair lock might be used to protect a shared resource that is only accessed by a few threads.

synchronized keyword

the synchronized keyword in Java is an unfair lock.

This means that when multiple threads are waiting to acquire a lock on a synchronized block,

the thread that gets the lock is not guaranteed to be the thread that has been waiting the longest.

Difference between fair locks and unfair locks

Feature	Guarantees order of lock acquisition	Typically used for
Fair lock	Yes	Situations where fairness is important
Unfair lock	No	Situations where performance is more important

Mutex lock

Mutex lock in java

A mutex, short for mutual exclusion, is a lock that is used to ensure that only one thread can access a shared resource at a time.

In Java, mutexes are implemented using the java.util.concurrent.locks.Lock interface.

synchronized keyword

The synchronized keyword in Java is a mutex lock. A mutex lock is a type of lock that allows only one thread to access a shared resource at a time.

The synchronized keyword ensures that only one thread can execute a synchronized block of code at a time.

This prevents race conditions and ensures that the shared resource is accessed in a consistent manner.

Renentrant lock

Purpose

A Renentrant Lock is a mutual exclusion mechanism that allows threads to reenter into a lock on a resource (multiple times) without a deadlock situation.

A thread entering into the lock increases the hold count by one every time. Similarly, the hold count decreases when unlock is requested.

Renentrant Locks in java

ReentrantLock and synchronized,

ReentrantLock allows a thread to acquire the same lock multiple times without blocking itself. This is known as reentrancy.

ReentrantLock also supports fairness, which means that threads that have been waiting for the lock the longest will be the first to acquire it.

synchronized block are reentrant in nature

i.e if a thread has lock on the monitor object and if another synchronized block requires to have the lock on the same monitor object,

then thread can enter that code block.

synchronized also does not support fairness.

Mutual exclusion

Yes, a ReentrantLock is a mutual exclusion (mutex).

Deadlock

Scenario

A deadlock in Java is a situation where two or more threads are blocked forever, waiting for each other.

```
Thread 1                Thread 2
   |                        |
   |                        |
  (Holding)               (Holding)
 Resource A             Resource B
   |                        |
   ▼                        ▼
(Requesting)          (Requesting)
 Resource B             Resource A
```

There are four conditions that must be met for a deadlock to occur:

- Mutual exclusion: Each resource must be held by at most one thread at a time.
- Hold and wait: A thread must be holding at least one resource while waiting for another resource.
- No preemption: A resource cannot be forcibly taken away from a thread.
- Circular wait: There must be a chain of two or more threads, each of which is waiting for a resource held by the next thread in the chain.

If all four of these conditions are met, then a deadlock can occur.

Avoidance

- Using timeouts:
  If a thread is waiting for a resource for too long, it should give up and try again later.
- Using lock hierarchies:
  If you have multiple resources that need to be locked, lock them in a specific order to avoid deadlocks.
  Using deadlock detection and avoidance algorithms:
  There are a number of algorithms that can be used to detect and avoid deadlocks.

Does deadlock and mutual exclusion

Deadlock does not only happen in mutual exclusion in Java.

Deadlock can occur in any situation where multiple threads are competing for resources and each thread is waiting for a resource that is held by another thread.

Spinlock

Purpose

A spinlock is a synchronization mechanism used in multithreaded programming to protect shared resources from being accessed by multiple threads at the same time.

Spinlocks are typically implemented using a busy-waiting loop,

where a thread that attempts to acquire a lock that is already held by another thread will repeatedly check the lock status until it becomes available.

Advantages and Disadvantages

Advantages:

- Spinlocks are very efficient when contention is low.
- Spinlocks are easy to implement.
- Spinlocks do not require any context switching.

Disadvantages:

- Spinlocks can lead to performance problems if the lock is held for a long period of time.
- Spinlocks can be unfair, as a thread that is spinning on a lock may be starved by other threads.
- Spinlocks can be difficult to debug.

Context Switching

When a thread tries to acquire a spinlock, it will keep trying until it is successful.

This means that the thread will not be put to sleep, and the operating system will not have to switch to another thread.

Implementation

In Java, spinlocks can be implemented using the AtomicBoolean class.

```
public class Spinlock {
  private AtomicBoolean locked = new AtomicBoolean(false);
  public void lock() {
    while (locked.getAndSet(true)) {
      // Busy-wait
    }
  }
  public void unlock() {
    locked.set(false);
  }
}
```

Adaptive Spinning and Lock Elimination

Adaptive Spinning

In Java, adaptive spinning is an optimization technique that can be used to improve the performance of lock-based synchronization.

When a thread attempts to acquire a lock that is already held by another thread, it can either spin or block.
Spinning means that the thread will continue to try to acquire the lock until it is successful.
Blocking means that the thread will suspend execution until the lock is released.

Adaptive spinning works by initially having the thread spin for a short period of time before blocking.
If the thread is successful in acquiring the lock during the spin period, then it avoids the overhead of blocking and context switching.
If the thread is not successful in acquiring the lock during the spin period, then it blocks.

Lock Elimination

Lock elimination is another optimization technique that can be used to improve the performance of lock-based synchronization.

Lock elimination works by eliminating the need for a lock altogether in certain cases.
For example, if a thread is the only thread that can access a particular piece of data, then there is no need for a lock to protect that data.

Lock elimination can be implemented using a number of different techniques.
One common technique is to use a technique called thread-local storage.

Thread-local storage allows each thread to have its own private copy of a variable.
If a thread is the only thread that can access a particular piece of data, then the data can be stored in thread-local storage.
This eliminates the need for a lock to protect the data.

Usage

Adaptive spinning is enabled by default in the HotSpot JVM.
The amount of time that a thread spins before blocking can be controlled using the -XX:AdaptiveSpinDuration JVM option.
Lock elimination is not enabled by default in the HotSpot JVM.
Lock elimination can be enabled using the -XX:+EliminateLocks JVM option.

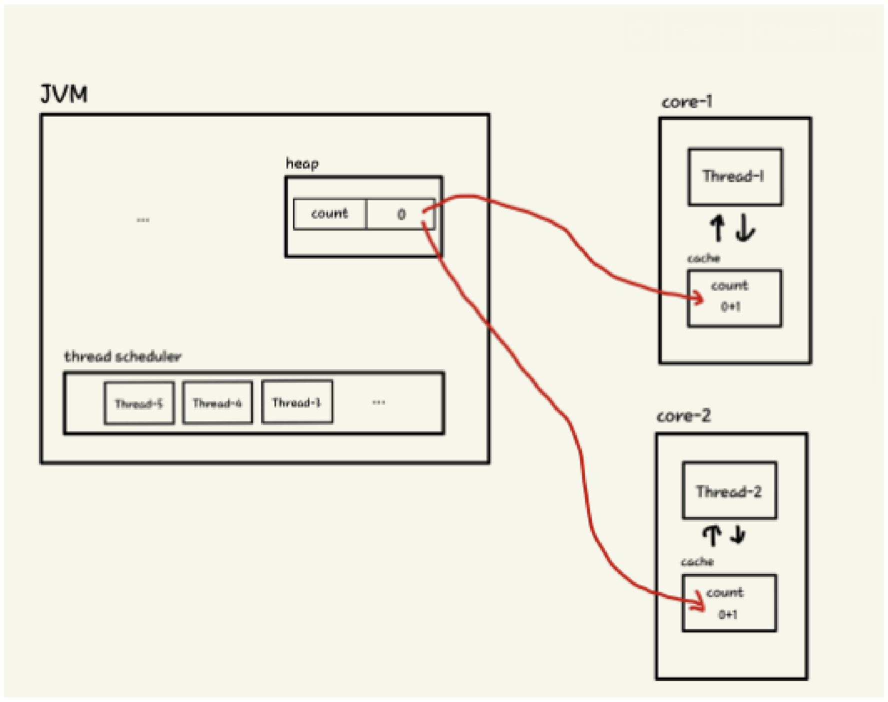
CAS

CAS

Compare and swap (CAS) is a concurrency control mechanism that allows a thread to atomically compare the value

of a memory location with a given value ,
and if the values match, modify the contents to a new given value.
(the "atomically" means that the entire operation cannot be interrupted and must be completed before any other thread can interfere)
CAS is used to implement synchronization primitives like semaphores and mutexes, as well as more sophisticated lock-free and wait-free algorithms.

When modifying shared data, save the original old value to the CPU cache.
First modify the value of the CPU cache variable.
When the memory value is about to be written, compare whether the cached old value and the memory value are equal to determine whether other threads have modified it.



Use CAS in JAVA

In Java, the compareAndSet() method of the AtomicInteger class implements CAS for primitive int variables.
The compareAndSet() method takes three parameters: the memory location to be updated, the expected value, and the new value.
If the value of the memory location matches the expected value, the compareAndSet() method updates the memory location to the new value and returns true.
Otherwise, the compareAndSet() method does not update the memory location and returns false.

```
AtomicInteger counter = new AtomicInteger(0);

// Increment the counter atomically.
boolean success = counter.compareAndSet(0, 1);

// If the compareAndSet() method was successful, the counter will now be 1.
// Otherwise, the counter will still be 0.
```

ABA problem

The ABA problem in computer science is a false positive execution of a CAS-based speculation on a shared location.
ABA is not an acronym and is a shortcut for stating that a value at a shared location can change from A to B and then back to A.

The ABA problem occurs when multiple threads (or processes) accessing shared data interleave.
The first thread assumes that nothing has happened in the interim,
But between the two reads there could be another thread changing the value A to B and then changing the value B to A.
(after reading the old value and before attempting to execute the CAS instruction).
Even if the first thread obtains A, it is no longer the previous A.
This can fool the first thread into thinking nothing has changed.

Java Lock Classification

Bias lock

Purpose

Biased locking in Java is specifically used to optimise the synchronized keyword.
Biased locking works when a method is not very concurrent, and only one thread usually acquires a lock.

Execution Process:

- The JVM raises a flag in the monitor object that some thread has acquired the lock, making it lightweight for the same thread to reacquire and release the lock.
  With biased locking, the first time a thread synchronizes on an object, it does a bit more work to acquire synchronized ('bias' it to the thread).
  Subsequent synchronizations proceed via a simple read test with no need to drain to cache.
  This makes subsequent monitor-related operations performed by that thread relatively much faster on multiprocess machines.
- However, the lock must be revoked when another thread tries to acquire the bias lock, which is a costly operation.

Biased locking is on by default in Java SE 6, and is disabled by default in Java SE 15 and slated for removal.
In Java SE 17, bias locking was deprecated and removed. This was due to a number of factors, including the fact that bias locking could lead to performance regressions in some cases.

Lightweight locks and heavyweight locks

Lightweight Lock

Lightweight locks are implemented using a technique called biased locking.
When a thread acquires a lightweight lock, the JVM sets a flag in the object header to indicate that the lock is owned by that thread.
If the same thread tries to acquire the lock again, the JVM can simply check the flag and grant the lock without having to perform any further synchronization.

However, if a different thread tries to acquire the lock, the JVM will have to perform a more expensive synchronization operation.
This is because the JVM needs to ensure that the other thread does not modify the object while the first thread is holding the lock.

Hightweight Lock

Heavyweight locks are implemented using a monitor.
A monitor is a data structure that contains a queue of threads that are waiting to acquire the lock.
When a thread acquires a heavyweight lock, it is added to the end of the queue.
When the thread that owns the lock releases it, the JVM wakes up the first thread in the queue and grants it the lock.

Difference

Lightweight locks and heavyweight locks are two different types of locks that can be used in Java to synchronize threads.

Heavyweight locks are less efficient than lightweight locks because they require the JVM to perform more synchronization operations.
However, heavyweight locks can be used in any situation,
while lightweight locks can only be used in situations where the same thread is likely to acquire the lock multiple times.
Here is a table summarizing the key differences between lightweight locks and heavyweight locks:

Feature	Lightweight lock	Heavyweight lock
Efficiency	More efficient	Less efficient
Applicability	Can only be used in certain situations	Can be used in any situation
Implementation	Uses biased locking	Uses a monitor

Segment Lock

Purpose

A segment lock in Java is a locking mechanism that is used to control access to a segment of data in a concurrent hash map.
A concurrent hash map is a data structure that allows multiple threads to access and modify the same data at the same time.
Segment locks are used to prevent conflicts between threads when they are trying to access the same segment of data.

Implementation

To implement a segment lock, the concurrent hash map is divided into a number of segments.
Each segment is associated with a lock.
When a thread wants to access a segment of data, it must first acquire the lock for that segment.
Once the thread has acquired the lock, it can access the data in the segment without interference from other threads.
When the thread is finished accessing the data, it must release the lock.

Here is an example of how to use a segment lock in Java:

```
import java.util.concurrent.ConcurrentHashMap;
public class Example {
  public static void main(String[] args) {
    ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

    // Acquire the lock for the segment that contains the key "key1".
    map.lock("key1");

    // Get the value associated with the key "key1".
    Integer value = map.get("key1");

    // Update the value associated with the key "key1".
    value++;
    map.put("key1", value);

    // Release the lock for the segment that contains the key "key1".
    map.unlock("key1");
  }
}
```

Lock Optimization

Reduce the lock granularity.

This means using smaller locks that only protect the specific data that needs to be protected.

For example, instead of locking an entire object, you could lock only the field that needs to be updated.

Coarsen locks

Reduce locking overhead by merging adjacent, consecutive lock operations.

This means moving the lock outside of a loop, so that it is only acquired once.

For example, instead of locking the list inside of a loop, you could lock the list before the loop and unlock it after the loop.

Reduce the lock holding time.

This means releasing the lock as soon as possible.

For example, instead of holding the lock while you are reading data, you could release the lock after you have read the data.

Use JDK-optimized concurrency classes.

The Java Development Kit (JDK) provides a number of concurrency classes that are optimized for performance.

For example, the ReentrantLock class is a re-entrant lock that can be used to protect shared data.

Class Loader

Constant pool:

Reference:

https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html#jvms-5.1

Classloader

Reference:

All about Java class loaders

https://www.infoworld.com/article/3700054/all-about-java-class-loaders.html

Class Loaders in Java

https://www.baeldung.com/java-classloaders

Java Virtual Machine Specification

https://docs.oracle.com/javase/specs/jvms/se7/html/jvms-5.html

Concept

Definition

In Java, a class loader is a mechanism that loads classes into the Java Virtual Machine (JVM).

It is responsible for finding, loading, and linking classes. The class loader is a part of the Java Runtime Environment (JRE) and is responsible for loading Java classes dynamically during runtime.

Load Timing

- When a class is referenced in a new expression (create an object).

- When a class is referenced in a static variable.

  (Except for the constants in the constant pool, because they have been put into the constant pool during the compilation process,

  using these constants does not directly refer to the current class.)
- When a class is referenced in a static method call.

- When a class is referenced in the instanceof operator.
- When a class is referenced in the cast operator.

- When a class is referenced by reflection.

  (Obtaining a Class object through the class name does not trigger class initialization.)

(When Class.forName method manually specifies the parameter "initialize" as false, initialization will not be executed.)

(ClassLoader.loadClass deafults to the second parameter false, and linking and initialization will not be executed.)

- When the main class is executed.

Startup

The Java Virtual Machine starts up by creating an initial class, which is specified in an implementation-dependent manner, using the bootstrap class loader.

The Java Virtual Machine then links the initial class, initializes it, and invokes the `public class method void main(String[])`.

The invocation of this method drives all further execution.

Execution of the Java Virtual Machine instructions constituting the main method may cause linking (and consequently creation) of additional classes and interfaces, as well as invocation of additional methods.

Custom Classloader

There are two kinds of class loaders:

the bootstrap class loader supplied by the Java Virtual Machine, and user-defined class loaders.

Every user-defined class loader is an instance of a subclass of the abstract class ClassLoader.

User-defined class loaders can be used to create classes that originate from user-defined sources.

For example, a class could be downloaded across a network, generated on the fly, or extracted from an encrypted file.

Load classes from a specific directory:

```
public class MyClassLoader extends ClassLoader {
    private String directory;
    public MyClassLoader(String directory) {
        this.directory = directory;
    }
    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        byte[] bytes = loadClassData(name);
        return defineClass(name, bytes, 0, bytes.length);
    }
    private byte[] loadClassData(String name) throws ClassNotFoundException {
        File file = new File(directory, name + ".class");
        if (!file.exists()) {
            throw new ClassNotFoundException("Class not found: " + name);
        }
        try {
            FileInputStream fis = new FileInputStream(file);
            byte[] bytes = new byte[(int) file.length()];
            fis.read(bytes);
            fis.close();
            return bytes;
        } catch (IOException e) {
            throw new ClassNotFoundException("Error loading class: " + name, e);
        }
    }
}
MyClassLoader classLoader = new MyClassLoader("/path/to/classes");
Class<?> myClass = classLoader.loadClass("MyClass");
```

Load classes from a jar file:

```
public class MyClassLoader extends ClassLoader {
    private URL jarFileURL;
    public MyClassLoader(URL jarFileURL) {
        this.jarFileURL = jarFileURL;
    }
    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
            try {
                JarURLConnection jarURLConnection = (JarURLConnection) jarFileURL.openConnection();
                JarFile jarFile = jarURLConnection.getJarFile();
                ZipEntry zipEntry = jarFile.getEntry(name + ".class");
                if (zipEntry == null) {
                    throw new ClassNotFoundException(name);
                }
                InputStream inputStream = jarFile.getInputStream(zipEntry);
                byte[] bytes = new byte[(int) zipEntry.getSize()];
                inputStream.read(bytes);
                inputStream.close();
                Class<?> clazz = defineClass(name, bytes, 0, bytes.length);
                return clazz;
            } catch (IOException e) {
                throw new ClassNotFoundException(name, e);
            }
        }
    }
    MyClassLoader myClassLoader = new MyClassLoader(new URL("file:///path/to/jar/file.jar"));
    Class<?> clazz = myClassLoader.loadClass("com.example.MyClass");
```

Classloader Types

Bootstrap ClassLoader:

Load core Java classes

Also known as the primordial class loader, this is the class loader where the search starts.
The bootstrap class loader is responsible for loading core Java classes such as java.lang.Object and java.lang.String.
It is implemented in native code and classes are located in the $JAVA_HOME/lib directory.

JDK 9

There were some important changes to class loaders between Java 8 and Java 9.
For example, in Java 8, the bootstrap class loader was located in the Java Runtime Environment's rt.jar file. In Java 9 and subsequently, the rt.jar file was removed.
Moreover, Java 9 introduced the Java module system, which changed how classes are loaded.
In the module system, each module defines its own class loader, and the bootstrap class loader is responsible for loading the module system itself and the initial set of modules.
When the JVM starts up, the bootstrap class loader loads the java.base module, which contains the core Java classes and any other modules that are required to launch the JVM.
The java.base module also exports packages to other modules, such as java.lang, which contains core classes like Object and String. These packages are then loaded by the bootstrap class loader.

Get and print bootstrap classloader:

```
public class BootstrapClassLoaderExample {
    public static void main(String[] args) {
        // Get the class loader for the String class, loaded by the Bootstrap Class Loader
        ClassLoader loader = String.class.getClassLoader();
        // Print the class loader's name
        System.out.println("Class loader for String class: " + loader);
    }
}
```

Output:

Class loader for String class: null

This is because the bootstrap class loader is the primordial class loader and does not have a name.

Extension ClassLoader

Load classes from the extension directory

It was used in earlier versions of Java to load classes from the extension directory, which was typically located in the JRE/lib/ext directory.
In Java 8, the extension classloader is known as the platform class loader.

JDK 9

In Java 9 and later versions, the extension class loader was removed from the JVM.

Instead of the extension class loader, Java 9 and later versions use the java.lang.ModuleLayer class to load modules from the extension directory.

The extension directory is now treated as a separate layer in the module system, and modules in the extension directory are loaded by the extension layer's class loader.

how to get and print extension classloader in java 8?

```
public class GetExtensionClassLoader {
    public static void main(String[] args) {
        ClassLoader extensionClassLoader = ClassLoader.getPlatformClassLoader();
        System.out.println("Extension classloader: " + extensionClassLoader);
    }
}
```

Output:

```
Extension classloader: sun.misc.Launcher$ExtClassLoader@7852e922
```

how to use the ModuleLayer class to load classes in java 9?

```
    ModuleLayer layer = ModuleLayer.boot();
    Module module = layer.findModule("java.base");
    ClassLoader classLoader = module.getClassLoader();
    Class<?> clazz = classLoader.loadClass("java.lang.String");
```

System ClassLoader

This ClassLoader loads classes from the classpath.

The classpath is a list of directories and JAR files that the JVM searches for classes. The classpath can be set using the CLASSPATH environment variable.

Process

Class loaders follow the delegation model,

where on request to find a class or resource, a ClassLoader instance will delegate the search of the class or resource to the parent class loader.

Let's say we have a request to load an application class into the JVM.

The system class loader first delegates the loading of that class to its parent extension class loader, which in turn delegates it to the bootstrap class loader.

Only if the bootstrap and then the extension class loader are unsuccessful in loading the class, the system class loader tries to load the class itself.

If the class is not found, the JVM throws a ClassNotFoundException.

Custom Class Loader

To implement a custom class loader, you need to extend the ClassLoader class and override the findClass method.

The findClass method is used to define how the class bytes are retrieved and converted into a Class object. Here's a basic example:

```
public class CustomClassLoader extends ClassLoader {

    // Constructor
    public CustomClassLoader(ClassLoader parent) {
        super(parent);
    }

    @Override
    protected Class<?> findClass(String name) throws ClassNotFoundException {
        // Convert the class name to the path of the class file
        String classPath = name.replace('.', '/').concat(".class");

        // Load the class file as a byte array
        byte[] classData = loadClassData(classPath);

        if (classData == null) {
            throw new ClassNotFoundException(name);
        }

        // Define the class with the byte array
        return defineClass(name, classData, 0, classData.length);
    }

    private byte[] loadClassData(String classPath) {
        // Implement loading of class data (byte array) from classPath
        // This could involve reading from a file system, network, etc.
        return null;
    }
}
```

Why Use loadClass Instead of findClass

In Java, the ClassLoader class has two main methods for class loading: loadClass and findClass.

findClass:

Purpose:

This is an abstract method in ClassLoader that you must override when you create a custom class loader.

Function:

It is responsible for locating and loading the class bytes. The method is only invoked by loadClass if the class is not already loaded.

loadClass:

Purpose:

This is the public method that is used to load classes by their name.

Function:

It handles the class loading process and calls findClass to delegate the actual class loading to the custom logic you provide.

It also manages class loading policies, such as ensuring that classes are loaded only once and delegating to parent class loaders.

Loading, linking and initialization

Loading

1. What is the loading process of classloader?

   Creation of a class or interface C denoted by the name N consists of the construction in the method area of the Java Virtual Machine (§2.5.4) of an implementation-specific internal representation of C.

   Class or interface creation is triggered by another class or interface D, which references C through its run-time constant pool.

   Class or interface creation may also be triggered by D invoking methods in certain Java SE platform class libraries (§2.12) such as reflection.

2. How is Array class loaded?

   If C is not an array class, it is created by loading a binary representation of C (§4) using a class loader.

   Array classes do not have an external binary representation; they are created by the Java Virtual Machine rather than by a class loader.

3. How does class loader load classes?

   A class loader L may create C by defining it directly or by delegating to another class loader.

If L creates C directly, we say that L *defines* C or, equivalently, that L is the *defining loader* of C.

When one class loader delegates to another class loader, the loader that initiates the loading is not necessarily the same loader that completes the loading and defines the class.

If L creates C, either by defining it directly or by delegation, we say that L initiates loading of C or, equivalently, that L is an *initiating loader* of C.
At run time, a class or interface is determined not by its name alone, but by a pair: its binary name (§4.2.1) and its defining class loader.
Each such class or interface belongs to a single *run-time package*.
The run-time package of a class or interface is determined by the package name and defining class loader of the class or interface.

The Java Virtual Machine uses one of three procedures to create class or interface C denoted by N:

- If N denotes a nonarray class or an interface, one of the two following methods is used to load and thereby create C:
    - If D was defined by the bootstrap class loader, then the bootstrap class loader initiates loading of C (§5.3.1).
    - If D was defined by a user-defined class loader, then that same user-defined class loader initiates loading of C (§5.3.2).
- Otherwise N denotes an array class. An array class is created directly by the Java Virtual Machine (§5.3.3), not by a class loader. However, the defining class loader of D is used in the process of creating array class C.

Linking

1. What is the linking process of classloader?

    Linking a class or interface involves verifying and preparing that class or interface, its direct superclass, its direct superinterfaces, and its element type (if it is an array type), if necessary.
    Resolution of symbolic references in the class or interface is an optional part of linking.

    This specification allows an implementation flexibility as to when linking activities (and, because of recursion, loading) take place,
    provided that all of the following properties are maintained:

    - A class or interface is completely loaded before it is linked.
    - A class or interface is completely verified and prepared before it is initialized.
    - Errors detected during linkage are thrown at a point in the program where some action is taken by the program that might, directly or indirectly, require linkage to the class or interface involved in the error.

2. What are the resolution strategies?

    For example, a Java Virtual Machine implementation may choose to resolve each symbolic reference in a class or interface individually when it is used ("lazy" or "late" resolution),
    or to resolve them all at once when the class is being verified ("eager" or "static" resolution).

    This means that the resolution process may continue, in some implementations, after a class or interface has been initialized.
    Whichever strategy is followed, any error detected during resolution must be thrown at a point in the program that (directly or indirectly) uses a symbolic reference to the class or interface.

HotSpot JVM

The HotSpot JVM resolves symbolic references on a lazy or on-demand basis.

This means it resolves each symbolic reference individually when it is first accessed, rather than resolving all symbolic references at once when the class is loaded.

3. Verification, Preparation, Resolution

Because linking involves the allocation **of new data structures, it may fail with an OutOfMemoryError.**

a) Verification

Verification (§4.10) ensures that the binary representation of a class or interface is structurally correct (§4.9).

Verification may cause additional classes and interfaces to be loaded (§5.3) but need not cause them to be verified or prepared.

If the binary representation of a class or interface does not satisfy the static or structural constraints listed in §4.9,

then a VerifyError must be thrown at the point in the program that caused the class or interface to be verified.

If an attempt by the Java Virtual Machine to verify a class or interface fails because an error is thrown that is an instance of LinkageError (or a subclass),

then subsequent attempts to verify the class or interface always fail with the same error that was thrown as a result of the initial verification attempt.

b) Preparation

Preparation involves creating the static fields for a class or interface and initializing such fields to their default values (§2.3, §2.4).

This does not require the execution of any Java Virtual Machine code; explicit initializers for static fields are executed as part of initialization (§5.5), not preparation.
During preparation of a class or interface C, the Java Virtual Machine also imposes loading constraints (§5.3.4).
Let L_1 be the defining loader of C. For each method `m` declared in C that overrides (§5.4.5) a method declared in a superclass or superinterface <D, L_2>, the Java Virtual Machine imposes the following loading constraints:

Given that the return type of `m` is T_r, and that the formal parameter types of `m` are T_{f1}, ..., T_{fn}, then:

If T_r not an array type, let T_0 be T_r; otherwise, let T_0 be the element type (§2.4) of T_r.

For *i* = 1 to *n*: If T_{fi} is not an array type, let T_i be T_{fi}; otherwise, let T_i be the element type (§2.4) of T_{fi}.

Then $T_i^{L_1} = T_i^{L_2}$ for *i* = 0 to *n*.

Furthermore, if C implements a method `m` declared in a superinterface <I, L_3> of C, but C does not itself declare the method `m`, then let <D, L_2> be the superclass of C that declares the implementation of method `m` inherited by C. The Java Virtual Machine imposes the following constraints:

Given that the return type of `m` is T_r, and that the formal parameter types of `m` are T_{f1}, ..., T_{fn}, then:

If T_r not an array type, let T_0 be T_r; otherwise, let T_0 be the element type (§2.4) of T_r.

For *i* = 1 to *n*: If T_{fi} is not an array type, let T_i be T_{fi}; otherwise, let T_i be the element type (§2.4) of T_{fi}.

Then $T_i^{L_2} = T_i^{L_3}$ for *i* = 0 to *n*.

Preparation may occur at any time following creation but must be completed prior to initialization.

c) Resolution

Symbolic References:

Initially, when a class is loaded, references to other classes, methods, and fields within the class file are symbolic. These references are not direct memory addresses but rather names or identifiers.

Runtime Resolution:

During the resolution phase, these symbolic references are replaced with direct references (e.g., memory addresses) to the actual classes, methods, or fields they point to.

On-Demand Resolution:

The resolution doesn't necessarily happen all at once. It can occur lazily, meaning that symbolic references are resolved as they are needed during execution.

The JVM no longer needs to resolve a constant pool reference if the specific reference is already resolved ahead-of-time (at dump time).

The Java Virtual Machine
instructions *anewarray*, *checkcast*, *getfield*, *getstatic*, *instanceof*, *invokedynamic*, *invokeinterface*, *invokespecial*, *invokestatic*, *invokevirtual*, *ldc*, *ldc_w*, *multianewarray*, *new*, *putfield*, and *putstatic*
make symbolic references to the run-time constant pool.
Execution of any of these instructions requires resolution of its symbolic reference.

Resolution is the process of dynamically determining concrete values from symbolic references in the run-time constant pool.
Resolution of the symbolic reference of one occurrence of an *invokedynamic* instruction *does not* imply that the same symbolic reference is considered resolved for any
other *invokedynamic* instruction.

For all other instructions above, resolution of the symbolic reference of one occurrence of an instruction does imply that the same symbolic reference is considered resolved for any other non-invokedynamic instruction.

(The above text implies that the concrete value determined by resolution for a
specific invokedynamic instruction is a call site object bound to that
specific invokedynamic instruction.)

Resolution can be attempted on a symbolic reference that has already been resolved.
An attempt to resolve a symbolic reference that has already successfully been resolved always succeeds trivially and always results in the same entity produced by the initial resolution of that reference.

If an error occurs during resolution of a symbolic reference, then an instance
of IncompatibleClassChangeError (or a subclass) must be thrown at a point in the program that (directly or indirectly) uses the symbolic reference.

If an attempt by the Java Virtual Machine to resolve a symbolic reference fails because an error is thrown that is an instance of LinkageError (or a subclass), then subsequent attempts to resolve the reference always fail with the same error that was thrown as a result of the initial resolution attempt.

A symbolic reference to a call site specifier by a specific invokedynamic instruction must not be resolved prior to execution of that instruction.

In the case of failed resolution of an invokedynamic instruction, the bootstrap method is not re-executed on subsequent resolution attempts.

Certain of the instructions above require additional linking checks when resolving symbolic references. For instance, in order for a getfield instruction to successfully resolve the symbolic reference to the field on which it operates, it must not only complete the field resolution steps given in §5.4.3.2 but also check that the field is not static. If it is a static field, a linking exception must be thrown.

Notably, in order for an invokedynamic instruction to successfully resolve the symbolic reference to a call site specifier, the bootstrap method specified therein must complete normally and return a suitable call site object. If the bootstrap method completes abruptly or returns an unsuitable call site object, a linking exception must be thrown.

Linking exceptions generated by checks that are specific to the execution of a particular Java Virtual Machine instruction are given in the description of that instruction and are not covered in this general discussion of resolution. Note that such exceptions, although described as part of the execution of Java Virtual Machine instructions rather than resolution, are still properly considered failures of resolution.

The following sections describe the process of resolving a symbolic reference in the run-time constant pool (§5.1) of a class or interface D. Details of resolution differ with the kind of symbolic reference to be resolved.

Initialization

1. What is the Initialization process of classloader?

   Initialization of a class or interface consists of executing its class or interface initialization method (§2.9).
   A class or interface may be initialized only as a result of:

   - The execution of any one of the Java Virtual Machine instructions new, getstatic, putstatic, or invokestatic that references the class or interface (§new, §getstatic, §putstatic, §invokestatic).
     All of these instructions reference a class directly or indirectly through either a field reference or a method reference.

     Upon execution of a new instruction, the referenced class or interface is initialized if it has not been initialized already.
     Upon execution of a getstatic, putstatic, or invokestatic instruction, the class or interface that declared the resolved field or method is initialized if it has not been initialized already.

   - The first invocation of a java.lang.invoke.MethodHandle instance which was the result of resolution of a method handle by the Java Virtual Machine (§5.4.3.5) and which has a kind of 2 (REF_getStatic), 4 (REF_putStatic), or 6 (REF_invokeStatic).
   - Invocation of certain reflective methods in the class library (§2.12), for example, in class Class or in package java.lang.reflect.
   - The initialization of one of its subclasses.
   - Its designation as the initial class at Java Virtual Machine start-up (§5.2).

   Prior to initialization, a class or interface must be linked, that is, verified, prepared, and optionally resolved.
   Because the Java Virtual Machine is multithreaded, initialization of a class ors interface requires careful

synchronization,
since some other thread may be trying to initialize the same class or interface at the same time.
There is also the possibility that initialization of a class or interface may be requested recursively as part of the initialization of that class or interface.
The implementation of the Java Virtual Machine is responsible for taking care of synchronization and recursive initialization by using the following procedure.
It assumes that the Class object has already been verified and prepared, and that the Class object contains state that indicates one of four situations:

- This Class object is verified and prepared but not initialized.
- This Class object is being initialized by some particular thread.
- This Class object is fully initialized and ready for use.
- This Class object is in an erroneous state, perhaps because initialization was attempted and failed.

For each class or interface C, there is a unique initialization lock LC. The mapping from C to LC is left to the discretion of the Java Virtual Machine implementation.
For example, LC could be the Class object for C, or the monitor associated with that Class object. The procedure for initializing C is then as follows:

1. Synchronize on the initialization lock, LC, for C. This involves waiting until the current thread can acquire LC.
2. If the Class object for C indicates that initialization is in progress for C by some other thread, then release LC and block the current thread until informed that the in-progress initialization has completed, at which time repeat this procedure.
3. If the Class object for C indicates that initialization is in progress for C by the current thread, then this must be a recursive request for initialization. Release LC and complete normally.
4. If the Class object for C indicates that C has already been initialized, then no further action is required. Release LC and complete normally.
5. **If the Class object for C is in an erroneous state, then initialization is not possible. Release LC and throw a NoClassDefFoundError.**
6. Otherwise, record the fact that initialization of the Class object for C is in progress by the current thread, and release LC. Then, initialize each final static field of C with the constant value in its ConstantValue attribute (§4.7.2), in the order the fields appear in the ClassFile structure.
7. Next, if C is a class rather than an interface, and its superclass SC has not yet been initialized, then recursively perform this entire procedure for SC. If necessary, verify and prepare SC first.

If the initialization of SC completes abruptly because of a thrown exception, then acquire LC, label the Class object for C as erroneous, notify all waiting threads, release LC, and complete abruptly, throwing the same exception that resulted from initializing SC.

8. Next, determine whether assertions are enabled for C by querying its defining class loader.
9. Next, execute the class or interface initialization method of C.
10. If the execution of the class or interface initialization method completes normally, then acquire LC, label the Class object for C as fully initialized, notify all waiting threads, release LC, and complete this procedure normally.
11. **Otherwise, the class or interface initialization method must have completed abruptly by throwing some exception E. If the class of E is not Error or one of its subclasses, then create a new instance of the class ExceptionInInitializerError with E as the argument, and use this object in place of E in the following step.**

If a new instance of ExceptionInInitializerError cannot be created because an OutOfMemoryError occurs, then use an OutOfMemoryError object in place of E in the following step.

12. Acquire LC, label the Class object for C as erroneous, notify all waiting threads, release LC, and complete this procedure abruptly with reason E or its replacement as determined in the previous step.

A Java Virtual Machine implementation may optimize this procedure by eliding the lock acquisition in step 1 (and release in step 4/5) when it can determine that the initialization of the class has already completed, provided that, in terms of the Java memory model, all happens-before orderings (JLS §17.4.5) that would exist if the lock were acquired, still exist when the optimization is performed.

2. What is the execution order of static blocks of parent class and subclass static in Java and the setting order of static fields?
   1) Static blocks of the parent class are executed.
   2) Static fields of the parent class are initialized.
   3) Static blocks of the child class are executed.
   4) Static fields of the child class are initialized.

   5) Instance blocks of the parent class are executed. （instantiation）
   6) Instance fields of the parent class are initialized.
   7) Constructor of the parent class is executed.
   8) Instance blocks of the child class are executed.
   9) Instance fields of the child class are initialized.
   10) Constructor of the child class is executed.

   Example:

```
class Parent {
    static {
        System.out.println("Parent static block");
    }


    static int parentStaticField = 10;


    {
        System.out.println("Parent instance block");
    }


    int parentInstanceField = 20;


    Parent() {
        System.out.println("Parent constructor");
    }
}


class Child extends Parent {
    static {
        System.out.println("Child static block");
    }


    static int childStaticField = 30;


    {
        System.out.println("Child instance block");
    }


    int childInstanceField = 40;


    Child() {
        System.out.println("Child constructor");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        new Child();
    }
}
```

Output:

```
Parent static block
Child static block
Parent instance block
Child instance block
Parent constructor
Child constructor
```

Java Reflection

Concept

1. What is the java reflection mechanism?

   Reflection is a feature in the Java programming language. It allows an executing Java program to examine or "introspect" upon itself,

   and manipulate internal properties of the program.

   For example, it's possible for a Java class to obtain the names of all its members and display them.

2. How to get the Class object use java reflection?
   - Using the Class.forName() method.
     ```
     Class<?> cls = Class.forName("com.example.MyClass");
     ```
   - Using the <obj>.getClass() method.
     ```
     MyClass obj = new MyClass();
     Class<?> cls = obj.getClass();
     ```
   - Using <class-name>.class.
     ```
     Class<?> cls = MyClass.class;
     ```
3. What is the difference between using "Class.forName" and "<class-name>.class" to get a class object?

   Obtaining a Class object through the class name does not trigger class initialization.

4. How to use reflection api to obtain information about fields, methods, constructors?

   Here is an example of how to use the Class object to get information about a class:
   ```
   Class<?> cls = Class.forName("com.example.MyClass");

   // Get the name of the class.
   String className = cls.getName();

   // Get the fields of the class.
   Field[] fields = cls.getDeclaredFields();

   // Get the methods of the class.
   Method[] methods = cls.getDeclaredMethods();

   // Get the constructors of the class.
   Constructor[] constructors = cls.getDeclaredConstructors();
   ```
   Here is an example of how to use the Class object to create a new instance of a class:
   ```
   Class<?> cls = Class.forName("com.example.MyClass");

   // Create a new instance of the class.
   Object obj = cls.newInstance();
   ```

Java EE Standard

Concept

1. What is java EE standard?

   Java Platform, Enterprise Edition (Java EE) is the standard in community-driven enterprise software.

Java EE is developed using the Java Community Process, with contributions from industry experts, commercial and open source organizations, Java User Groups, and countless individuals.

2. What does java SE stand for?

   SE stands for Java Standard Edition.

   Java SE is a computing platform that allows developers to build and deploy Java applications on servers and desktops. It's part of the Java software-platform family and uses the Java programming language. Java SE was previously known as Java 2 Platform, Standard Edition (J2SE).

3. Does java ee 6 include java ee 5 features?

   Yes, Java EE 6 includes some features from Java EE 5, along with new technologies and usability improvements.

Java EE 5

1. Web Services Technologies

Implementing Enterprise Web Services	JSR 109
Java API for XML-Based Web Services (JAX-WS) 2.0	JSR 224
Java API for XML-Based RPC (JAX-RPC) 1.1	JSR 101
Java Architecture for XML Binding (JAXB) 2.0	JSR 222
SOAP with Attachments API for Java (SAAJ)	JSR 67 JSR 173
Web Service Metadata for the Java Platform	JSR 181

2. Web Application Technologies

JavaServer Faces 1.2	JSR 252
JavaServer Pages 2.1	JSR 245
JavaServer Pages Standard Tag Library	JSR 52
Java Servlet 2.5	JSR 154

3. Enterprise Application Technologies

Common Annotations for the Java Platform	JSR 250
Enterprise JavaBeans 3.0	JSR 220
J2EE Connector Architecture 1.5	JSR 112
JavaBeans Activation Framework (JAF) 1.1	JSR 925
JavaMail	JSR 919
Java Message Service API	JSR 914
Java Persistence API	JSR 220
Java Transaction API (JTA)	JSR 907

4. Management and Security Technologies

J2EE Application Deployment	JSR 88
J2EE Management	JSR 77
Java Authorization Contract for Containers	JSR 115

Java EE 6

1. Web Services Technologies

Java API for RESTful Web Services (JAX-RS) 1.1	JSR 311
Implementing Enterprise Web Services 1.3	JSR 109
Java API for XML-Based Web Services (JAX-WS) 2.2	JSR 224
Java Architecture for XML Binding (JAXB) 2.2	JSR 222
Web Services Metadata for the Java Platform	JSR 181
Java API for XML-Based RPC (JAX-RPC) 1.1	JSR 101
Java APIs for XML Messaging 1.3	JSR 67
Java API for XML Registries (JAXR) 1.0	JSR 93

2. Web Application Technologies

Java Servlet 3.0	JSR 315

JavaServer Faces 2.0	JSR 314
JavaServer Pages 2.2/Expression Language 2.2	JSR 245
Standard Tag Library for JavaServer Pages (JSTL) 1.2	JSR 52
Debugging Support for Other Languages 1.0	JSR 45

3. Enterprise Application Technologies

Contexts and Dependency Injection for Java (Web Beans 1.0)	JSR 299
Dependency Injection for Java 1.0	JSR 330
Bean Validation 1.0	JSR 303
Enterprise JavaBeans 3.1 (includes Interceptors 1.1)	JSR 318
Java EE Connector Architecture 1.6	JSR 322
Java Persistence 2.0	JSR 317
Common Annotations for the Java Platform 1.1	JSR 250
Java Message Service API 1.1	JSR 914
Java Transaction API (JTA) 1.1	JSR 907
JavaMail 1.4	JSR 919

4. Management and Security Technologies

Java Authentication Service Provider Interface for Containers	JSR 196
Java Authorization Contract for Containers 1.3	JSR 115
Java EE Application Deployment 1.2	JSR 88
J2EE Management 1.1	JSR 77

5. Java EE-related Specs in Java SE

Java API for XML Processing (JAXP) 1.3	JSR 206
Java Database Connectivity 4.0	JSR 221
Java Management Extensions (JMX) 2.0	JSR 255
JavaBeans Activation Framework (JAF) 1.1	JSR 925
Streaming API for XML (StAX) 1.0	JSR 173

Java / Usage

Xml reading and writing

There are four main ways for JAVA to manipulate XML documents, namely DOM, SAX, JDOM, and DOM4J

The DOM and SAX is provided by official, and JDOM and DOM4J uses threeparty libraries,

DOM and SAX are officially provided, while JDOM and DOM4J refer to third-party libraries, with the most commonly used being the DOM4J method.

The best one in terms of operational efficiency and memory usage is SAX, but due to its event based approach, SAX cannot modify the written content during the process of writing XML .

But for requirements that do not require frequent modifications, SAX should still be chosen.

DOM

File file = new File(xmlSavePath + "/mb.xml");
File copyFile = new File(xmlSavePath + "/" + filename + ".xml"); Copy Template xml
FileUtils.copyFile(file, copyFile);

DocumentBuilderFactory documentBuilderFactory = DocumentBuilderFactory.newInstance();
DocumentBuilder documentBuilder = documentBuilderFactory.newDocumentBuilder(); Create a DocumentBuilder object to convert XML files into Document Objects
Document document = documentBuilder.parse(copyFile); Create a Document object to parse xml files.

Node processor = document.getElementsByTagName("PROCESSOR").item(0);   Get the value of the "PROCESSOR" tag.
NamedNodeMap processors = processor.getAttributes();
processors.getNamedItem("path").setTextContent(systemConfig.getAnalysisToolValus());   Modify the "path" attribute.

TransformerFactory transformerFactory = TransformerFactory.newInstance();
Transformer transformer = transformerFactory.newTransformer();
DOMSource domSource = new DOMSource(document);
StreamResult reStreamResult = new StreamResult(copyFile);
transformer.transform(domSource, reStreamResult);

SAX

SAX is an event-driven, stream-based XML parsing method. Unlike **DOM** (Document Object Model), SAX does **not** load the entire XML document into memory. Instead, it reads and processes the XML content sequentially, making it an excellent choice for parsing large XML files efficiently.

Key Differences

SAX1:

- Limited in functionality.
- Does **not** support separation of local names, qualified names (QName), and namespace URIs.

SAX2:

- Enhanced capabilities.
- Supports local names, QNames, and namespace URIs.

Custom SAX Handler Example

```
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class PomParseHandler extends DefaultHandler {
    @Override
    public void startElement(String uri, String localName, String qName, Attributes attributes) throws 
SAXException {
        System.out.println("Start Element: " + qName);
        // Process attributes if needed
        for (int i = 0; i < attributes.getLength(); i++) {
            System.out.println("Attribute: " + attributes.getQName(i) + " = " + attributes.getValue(i));
        }
    }

    @Override
    public void endElement(String uri, String localName, String qName) throws SAXException {
        System.out.println("End Element: " + qName);
    }

    @Override
    public void characters(char[] ch, int start, int length) throws SAXException {
        String content = new String(ch, start, length).trim();
        if (!content.isEmpty()) {
            System.out.println("Content: " + content);
        }
    }
}
```

Parsing XML with SAX

SAX Parser (Legacy)

```
// SAX Parser to read XML
SAXParserFactory factory = SAXParserFactory.newInstance();
```

```
SAXParser parser = factory.newSAXParser();
PersonHandler personHandler = new PersonHandler();
parser.parse(inStream, personHandler);
inStream.close();
return personHandler.getPersons();

SAX2 Parser (Modern)
// SAX2 Parser to read XML
PomParseHandler sax2Handler = new PomParseHandler();
XMLReader xmlReader = XMLReaderFactory.createXMLReader();
xmlReader.setContentHandler(sax2Handler);
xmlReader.setErrorHandler(sax2Handler);

FileReader fileReader = new FileReader("./src/sample.xml");
xmlReader.parse(new InputSource(fileReader));
```

Sample XML

<?xml version="1.0" encoding="utf-8"?>

<websites

  xmlns:sina="http://www.sina.com"

  xmlns:baidu="http://www.baidu.com">

  <sina:website sina:blog="blog.sina.com">Sina</sina:website>

  <baidu:website baidu:blog="hi.baidu.com">Baidu</baidu:website>

</websites>

Namespace Handling Comparison

SAX1

For the attribute sina:blog="blog.sina.com", the parsing result is:

- LocalName = "sina:blog"
- QName = "sina:blog"
- URI = ""
- Value = "blog.sina.com"

SAX2:=

For the same attribute, the parsing result is:

- LocalName = "blog"
- QName = "sina:blog"
- URI = "http://www.sina.com"
- Value = "blog.sina.com"

XPath

XPath Example

This example demonstrates how to use XPath in Java to extract all attributes (@*) from a specific XML node.

```
import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import org.xml.sax.InputSource;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathFactory;
import java.io.StringReader;

public class XPathAttributeExample {
    public static void main(String[] args) throws Exception {
        String xml = "<root><comment author=\"Alice\" date=\"2025-05-12\">Sample comment</comment></root>";
        // Step 1: Parse the XML into a Document
        DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
        Document document = builder.parse(new InputSource(new StringReader(xml)));

        // Step 2: Retrieve the <comment> element
        NodeList commentNodes = document.getElementsByTagName("comment");
        if (commentNodes.getLength() == 0) {
            System.out.println("No <comment> element found.");
            return;
        }

        // Step 3: Use XPath to get all attributes of the <comment> element
        XPathFactory xpathFactory = XPathFactory.newInstance();
        XPath xpath = xpathFactory.newXPath();

        NodeList attributeNodes = (NodeList) xpath.evaluate(
            "//@*",
            commentNodes.item(0),
            XPathConstants.NODESET
        );

        // Step 4: Print all attributes
        for (int i = 0; i < attributeNodes.getLength(); i++) {
            System.out.println(attributeNodes.item(i).getNodeName() + " = " +
                               attributeNodes.item(i).getNodeValue());
        }
    }
}
```

Java / Core

Constants and Variables

byte

Size:

1 byte (8 bits)

Range:

From -128 to 127

Example:

```
byte a = 100;                // Decimal
byte a=127,b=a/10;          # Multiple assignment
byte a = 0b1100100;         // Binary (100 in decimal)
byte a = 0x64;              // Hexadecimal (100 in decimal)
byte a=-017;
byte a='c'
```

short

Size:

2 bytes (16 bits)

Range:

From -32,768 to 32,767

Example:

```
short a = 999;              // Decimal
short a=999, b=a/10;        // Multiple assignment
short a = 0b1000;           // Binary (8 in decimal)
short a=-017;                  // Octal (-15 in decimal)
short a=-0x189;                  // Hexadecimal (-393 in decimal)
```

int

Size:

Typically 4 bytes (32 bits)

Range:

From -2^31 to 2^31 - 1 (i.e., -2,147,483,648 to 2,147,483,647)

Example:

```
int a = 999;                // Decimal
int a, b = 100, 200;     // Multiple assignment
int a = 0b1000;              // Binary (8 in decimal)
int a = -017;                 // Octal (-15 in decimal)
int a = 0x189;               // Hexadecimal (393 in decimal)
```

long

Size:

Typically 8 bytes (64 bits)

Range:

From -2^63 to 2^63 - 1

Example:

```
long a = 88_999L;                // Decimal
long a, b = 100, 200;     // Multiple assignment
long a = 0b1000L;              // Binary (8 in decimal)
long a = -017l;                 // Octal (-15 in decimal)
long a = 0x189L;               // Hexadecimal (393 in decimal)
```

float

Size:

Typically 4 bytes (32 bits)

Range:

Approximately ±1.4 × 10^(-45) to ±3.4 × 10^(38)

Example:

```
float a=3;                        // Decimal
float a = 3.0f + 4.0f;         // Complex with float parts
float a =.75f;
float a=3.88f;
float a = 0b1000f;            // Binary (8 in decimal)
float a = 0x189p0f;           // Hexadecimal (393 in decimal)
float a=4.2F;
float a=2e-5f;
float a=4.4E-10F;
```

double

Size:

Typically 8 bytes (64 bits)

Range:

Approximately ±1.7 × 10^(-308) to ±1.7 × 10^(308) (usually IEEE 754 double precision)

Loss of precision

The double data type in Java suffers from loss of precision due to the way it represents numbers internally.
Unlike decimal numbers (base-10), double uses a binary format (base-2) with a limited number of bits.
This means certain decimal values can't be perfectly represented in binary.

Example:

```
double a=3;                       // Decimal
double a =.75;
double a=3.88;
double a = 0b1000;          // Binary (8 in decimal)
double a = 0x189p0;          // Hexadecimal (393 in decimal)
double a=4.2d;
double a=2e-5D;
double a=4.4E-10;
```

char

The char type is equal to integer duiring calculation.
During calculations, the character type is equal to an integer.

ASCII: 48-56 (0-9)   65-90 (A-Z)   97-122 (a-z)

a - 32 can convert uppercase letters to lowercase.
a +32 can convert lowercase letters to uppercase.

Size:

2 bytes (16 bits)

Range:

From '\u0000' to '\uffff' (0 to 65,535)

Example:

```
char a='x';
char a=68;
char a='\u005d';
```

Usage:

```
Integer.valueOf('0')     //  48
(int)'0'                //  48
2+'0'                   //  50

(char)0                 //  ?     (wrong result)
(char) 65               //  A
(char) 48               //  0
(char)(2+'0')           //  2

Character.getNumericValue('2')   //  2
Character.getNumericValue('*')   //  -1

Character c='c'.
c.equals('c')
    // Compilation error, The equals() method is intended for comparing objects, not primitive types.
c=='c'   // true
```

boolean

Size:

1 byte (though JVM implementation may optimize storage)

Example:

```
boolean a=false;
boolean b=true;
```

String

Size:

Varies based on the content

Example:

```
String a = "Hello, World!";          // String literal
String a = new String("Hello");     // String object creation
String a = """   // Multi-line Strings (Java 15+)
content
""";
```

Concatenation Efficiency:

StringBuilder >> concat() > +

String Comparison:

Use .equals() instead of == to compare content, as == compares references (memory addresses).

Escape Characters:

Use double backslashes (\\) in Java to represent a single backslash.

Usage:

```
String result = "a" + null + "b";  // Output: "anullb"
```

Constants

final variable:

Value cannot change once assigned.

Example:

```
final int a;
```

final method:

Cannot be overridden.

Example:

```
final void handleGet(){}
```

final class:

Cannot be inherited.

Example:

```
final class Person {}
```

Array

Definition

Arrays must be initialized with a fixed size.

Default values for primitive arrays: 0 for int, 0.0 for double, false for boolean, etc.

Accessing out-of-bound indexes will throw ArrayIndexOutOfBoundsException.

Example:

```
int[] a = {1, 2, 3, 4};
int a[] = new int[]{1, 2, 3, 4, 5};
double[] a = new double[6];
int[] a = arr.clone();
Map<Integer, Integer>[] maps = new Map[k];
Set<Integer>[] g = new Set[5];
```

Wrapper Classes (Reference Types)

Boolean, Character, Byte, Short, Integer, Long, Float, Double

Reference types default to null if uninitialized.

Primitive types have specific default values (e.g., 0, false).

Types of Objects in Java

POJO (Plain Old Java Object)

A simple Java object with no specific constraints or rules. This category includes DTO, VO, BO, and PO.

DTO (Data Transfer Object)

An object used solely for transferring data. Some projects further divide it into InDTO and OutDTO, which represent input and output DTOs, respectively.

DAO (Data Access Object)

An object responsible for accessing the database. It performs database operations and returns the required data.

VO (Value Object)

A value object used as a transport object to carry data between layers or systems.

PO (Persistent Object)

A persistent object, typically representing a mapping to a database table.

BO (Business Object)

A business object designed to fulfill business requirements. It may consist of multiple POs or even be a single PO.

It is often used as a transport object to display data to the frontend or provide data to an external system by combining multiple POs to meet the requirements.

Types of References in Java

Strong References

This is the default type/class of Reference Object.

Any object which has an active strong reference are not eligible for garbage collection.

The object is garbage collected only when the variable which was strongly referenced points to null.

Strength Rank:

Strong References > Soft References > Weak References > Phantom References

Soft References

Definition

A type of reference that allows an object to be garbage collected only if the JVM is low on memory.

Characteristics:

- Retention

  Softly reachable objects remain in memory as long as there is no memory pressure.
- Garbage Collection

  Soft references are collected less aggressively than weak references.

  The JVM will clear soft references before throwing an OutOfMemoryError.
- Use Case

  Ideal for implementing memory-sensitive caches where you want to retain objects in memory as long as possible but can release them when memory is needed.

Example:

```
import java.lang.ref.SoftReference;
class Gfg {
    public void x()
    {
        System.out.println("GeeksforGeeks");
    }
}

public class Example {
    public static void main(String[] args) {
        // Strong Reference
        Gfg g = new Gfg();
        g.x();
        // Creating Soft Reference to Gfg-type object to which 'g' is also pointing.
        SoftReference<Gfg> softref = new SoftReference<Gfg>(g);

        // Now, Gfg-type object to which 'g' was pointing earlier is available for garbage collection.
        g = null;

        // Simulate garbage collection
        System.gc();


        // You can retrieve back the object which has been weakly referenced.
        // It successfully calls the method.
        g = softref.get();

        g.x();
    }
}
```

Weak References

Definition

A type of reference that allows an object to be garbage collected at any time if there are no strong or soft references pointing to it.

Characteristics:

- Retention

  Weakly reachable objects can be collected at any point if there are no strong references to them.
- Garbage Collection

  Weak references are cleared more aggressively than soft references.

  They are typically collected in the next garbage collection cycle after becoming weakly reachable.
- Use Case

  Useful for implementing canonicalizing mappings, like WeakHashMap, where you want entries to be removed once they are no longer referenced elsewhere.

Example:

```
import java.lang.ref.WeakReference;
class Gfg {
    public void x()
    {
        System.out.println("GeeksforGeeks");
    }
}

public class Example {
    public static void main(String[] args) {
        // Strong Reference
        Gfg g = new Gfg();
        g.x();

        // Creating Weak Reference to Gfg-type object to which 'g' is also pointing.
        WeakReference<Gfg> weakref = new WeakReference<Gfg>(g);

        //Now, Gfg-type object to which 'g' was pointing earlier is available for garbage collection.
        //But, it will be garbage collected only when JVM needs memory.
        g = null;

        // You can retrieve back the object which has been weakly referenced.
         // It successfully calls the method.
        g = weakref.get();

        g.x();
    }
}
```

Phantom References

Definition

A type of reference used to schedule post-mortem cleanup actions after an object has been finalized but before its memory is reclaimed by the garbage collector.

Characteristics:

Enqueuing

Phantom references are enqueued in a reference queue after the object's finalization but before the object's memory is reclaimed.

(They are put in a reference queue after calling finalize() method on them)

Phantom references themselves do not provide a way to manually reclaim memory, but they allow you to perform cleanup operations before the memory is reclaimed by the garbage collector.

The key point is that phantom references can be used to execute actions after an object has been finalized but before the memory is reclaimed,

giving you an opportunity to manage resources like closing file handles, releasing native resources, etc.

Access

The get() method of a phantom reference always returns null.

Use Case

Typically used for pre-mortem cleanup operations, like deallocating native resources.

Example:

```
import java.lang.ref.*;
class Gfg {
    public void x() {
        System.out.println("GeeksforGeeks");
    }
}
public class ResourceCleaner {
    public static void cleanUp(Object obj) {
        // Code to clean up native resources
        System.out.println("Cleaning up resources for " + obj);
    }
}

public class Example {
    public static void main(String[] args)     {
        //Strong Reference
        Gfg g = new Gfg();
        g.x();

        //Creating reference queue
        ReferenceQueue<Gfg> refQueue = new ReferenceQueue<Gfg>();

        //Creating Phantom Reference to Gfg-type object to which 'g' is also pointing.
        PhantomReference<Gfg> phantomRef = null;

        phantomRef = new PhantomReference<Gfg>(g,refQueue);

        //Now, Gfg-type object to which 'g' was pointing earlier is available for garbage collection.
        //But, this object is kept in 'refQueue' before removing it from the memory.
        g = null;

        // Simulate garbage collection
        System.gc();

        //It always returns null.
        g = phantomRef.get();

        //It shows NullPointerException.
        g.x();
```

```
            // Poll the reference queue
            Reference<?> ref = refQueue.poll();
            if (ref != null) {
                ResourceCleaner.cleanUp(ref);
                // Perform other cleanup actions
            }

        }
    }
```

String Literals

Characteristics

Escape Sequence

Characters may be represented by escape sequences (§3.10.6) - one escape sequence for characters in the range U+0000 to U+FFFF,

two escape sequences for the UTF-16 surrogate code units of characters in the range U+010000 to U+10FFFF.

Unicode

Instances of *class* String represent sequences of Unicode code points.

A String object has a constant (unchanging) value.

String literals (§3.10.5) are references to instances of class String.

The string concatenation operator + (§15.18.1) implicitly creates a new String object when the result is not a compile-time constant expression (§15.28).

Compilation Process

Classes:

```
package testPackage;
class Test {
    public static void main(String[] args) {
        String hello = "Hello", lo = "lo";
        System.out.print((hello == "Hello") + " ");
        System.out.print((Other.hello == hello) + " ");
        System.out.print((other.Other.hello == hello) + " ");
        System.out.print((hello == ("Hel"+"lo")) + " ");
        System.out.print((hello == ("Hel"+lo)) + " ");
        System.out.println(hello == ("Hel"+lo).intern());
    }
}
class Other { static String hello = "Hello"; }
```

produces the output:

```
true true true true false true
```

This example illustrates six points:

- Literal strings within the same class (§8) in the same package (§7) represent references to the same String object (§4.3.1).
- Literal strings within different classes in the same package represent references to the same String object.
- Literal strings within different classes in different packages likewise represent references to the same String object.
- Strings computed by constant expressions (§15.28) are computed at compile time and then treated as if they were literals.
- Strings computed by concatenation at run time are newly created and therefore distinct.
- The result of explicitly interning a computed string is the same string as any pre-existing literal string with the same contents.

Widening or Automatic Type Conversion

Widening conversion takes place when two data types are automatically converted. This happens when:

- The two data types are compatible.
- When we assign a value of a smaller data type to a bigger data type.

  short, byte, char → int → long → float → double

  ```
  double i = 1/3.33;      //  1.00/3.33
  float i = 23;
  ```
- When the result may exceed the limits of the data type, relying on automatic type conversion can lead to precision loss.

  In such cases, explicit type casting should be used to prevent overflow:

  ```
  sum1-=(long) arr[preInd]*(preInd-stack.peek());
  ```

Narrowing or Explicit Conversion

If we want to assign a value of a larger data type to a smaller data type we perform explicit type casting or narrowing.

- This is useful for incompatible data types where automatic conversion cannot be done.
- Here, the target type specifies the desired type to convert the specified value to.

  (For primitive types, you can use Widening Type Conversion or Narrowing Type Conversion. but for objects, you can only use Narrowing Type Conversion to convert a subclass to a parent classes.)

  ```
  double d = 100.04;
  long l = (long)d;
  int i = (int)l;
  char c=(char)2

  Child child = new Child();
  Parent parent = (Parent)child;
  ```

Common Data Type Conversion

Convert string to other types

Integer.parseInt("22")

Converts a string to an integer.

Throws an exception if the string contains non-numeric characters or is improperly formatted (e.g., decimals or characters).

Byte.parseByte("33")

Short.parseShort("44")

Long.parseLong("55")

Float.parseFloat("99" or "99.8")

Double.parseDouble("99" or "99.8")

Convert float to other types

Float a= new Float("33" or "33.8" or 33 or 33.8)

Constructs a Float object from a string or primitive float.

The string can represent both integer and decimal values.

a.doubleValue()

a.intValue()

Convert double to other types

Double a= new Double("33" or "33.8" or 33 or 33.8)

a.intValue()

Convert enum to string

String.valueOf(RED)

Operators

Reference:

Guide to the Volatile Keyword in Java

https://www.baeldung.com/java-volatile

Keyword

instanceof

The instanceof keyword checks whether an object is an instance of a specific class or its subclass.

```
if (person instanceof Person) {
    // true if 'person' is an instance of Person or a subclass
}
```

out

Enhanced for Loop with Label

```
out: for (String val : arr) {
    // Use 'continue out;' or 'break out;' to control the outer loop
}
```

assert

Used for debugging purposes to make assumptions explicit.

```
assert listFiles != null : "listFiles should not be null";
```

try-catch-finally

```
public class Example {
    public static void main(String[] args) {
        System.out.println(testMethod());
    }

    public static String testMethod() {
        try {
            throw new Exception("An exception occurred");
        } catch (Exception e) {
            System.out.println("Caught exception: " + e.getMessage());
            return "Returning from catch block";
        } finally {
            System.out.println("Finally block executed");
        }
    }
}
```

Output:

```
Caught exception: An exception occurred
Finally block executed
Returning from catch block
```

IO Operations with Try-With-Resources

Automatically closes resources that implement the Closeable or AutoCloseable interfaces.

```
try (OutputStream out1 = new FileOutputStream("file1.txt");
     OutputStream out2 = new FileOutputStream("file2.txt")) {

    // Perform write operations
} catch (Exception e) {
    e.printStackTrace();
}
```

package and import

```
// Declare the package
package a;            // or: package a.b;

// Import statements
import a.b.*;         // Imports all classes in package a.b
import a.b.Tom;       // Imports only the class Tom from a.b

// Usage when class names conflict:
a.b.Tom obj = new a.b.Tom();
```

Synchronization Keyword

Reference:

https://www.baeldung.com/java-volatile

volatile

Thread and Process

Processes

A process is an independent entity, and its threads can run concurrently on multiple CPU cores.

The process itself does not directly interact with CPU cores but rather through its threads.

Threads

A single thread runs on one CPU core at a time. In a multi-core system,

different threads from the same or different processes can run concurrently on multiple cores.

CPU Core Optimization

Modern processors use several optimization techniques to improve performance and efficiency:

Caching

Each CPU core has its own cache (L1, L2, and sometimes L3) to store frequently accessed data and instructions,

significantly speeding up access compared to fetching from RAM.

Memory Visibility

Cache coherence primarily refers to the consistency between the caches of different cores, such as Core1 and Core2,

rather than the coherence within the caches of a single core.

```
public class TaskRunner {
  private static int number;
  private static boolean ready;
  private static class Reader extends Thread {
    @Override public void run() {
      while (!ready) {
        Thread.yield();
      }
      System.out.println(number);
    }
  }
  public static void main(String[] args) {
    new Reader().start();
    number = 42;
    ready = true;
  }
}
```

We may expect the reader thread to print 42 after a short delay.

however, the delay may be much longer. It may even hang forever.

Let's imagine a scenario in which the OS schedules those threads on two different CPU cores, where:

- The main thread has its copy of ready and number variables in its core cache.
- The reader thread ends up with its copies, too.
- The main thread updates the cached values. (the reader thread may immediately see the updated value, with some delay, or never at all.)

Out of Order Execution:

Allows the CPU to execute instructions out of their original order to utilize resources more effectively and avoid delays.

```
public static void main(String[] args) {
  new Reader().start();
  number = 42;
  ready = true;
}
```

We may expect the reader thread to print 42.

But it's actually possible to see zero as the printed value.

Branch Prediction:

Predicts the direction of branches (such as if-else statements) to minimize delays caused by branch instructions.

Speculative Execution:

Executes instructions before it is certain they are needed based on predicted outcomes to reduce wait times.

Volatile Memory Order

We can use volatile to tackle the issues with Cache Coherence.

To ensure that updates to variables propagate predictably to other threads, we should apply the volatile modifier to those variables.

This way, we can communicate with runtime and processor to not reorder any instruction involving the volatile variable.

Also, processors understand that they should immediately flush any updates to these variables.

```
public class TaskRunner {

    private volatile static int number;
    private volatile static boolean ready;

    // same as before
}
```

Happens-Before Relationship

The "happens-before" relationship guarantees that all memory writes that happen before the write to the volatile variable in one thread become visible to any thread that reads that volatile variable.

This is true regardless of whether the non-volatile variable is in the same cache line as the volatile variable.

Piggybacking

Because of the strength of the happens-before memory ordering, sometimes we can piggyback on the visibility properties of another volatile variable.

For instance, in our particular example, we just need to mark the ready variable as volatile:

```
public class TaskRunner {

    private static int number; // not volatile
    private volatile static boolean ready;

    // same as before
}
```

Anything prior to writing true to the ready variable is visible to anything after reading the ready variable.

Therefore, the number variable piggybacks on the memory visibility enforced by the ready variable.

Simply put, even though it's not a volatile variable, it's exhibiting a volatile behaviour.

Using the volatile Keyword Safely in Java

Volatile for Visibility, Not Thread Safety:

- While volatile ensures that the latest value is visible, it does not make operations like addition atomic.
- If multiple threads concurrently try to update the same volatile variable (e.g., incrementing it), a race condition can occur.

Avoid Non-Atomic Operations:

- Avoid using operations like i++ to update a volatile variable. The i++ operation involves both a read and a write, which are two separate operations.
- In a multi-threaded environment, this can lead to race conditions because the read and write are not atomic.
- Multiple threads might read the same value and then write back a result that doesn't account for other threads' operations.

False Sharing

False sharing occurs in a multi-threaded environment when multiple threads modify variables that reside on the same cache line, leading to unnecessary cache coherence traffic and performance degradation.
False sharing is not limited to volatile variables; it can happen with any variables that are accessed by multiple threads if those variables are located on the same cache line.

How It Happens:

- Cache Lines:
  - Modern processors use cache lines (typically 64 bytes) to read and write data from and to memory.
  - When a cache line is loaded into the CPU cache, all the data within that line is loaded.
- Variable Location:
  - If two or more variables used by different threads are located on the same cache line,
    modifications to any of these variables will cause the entire cache line to be invalidated and reloaded in other processors’ caches.
- Cache Coherence Overhead:
  - Even if the variables themselves do not interfere with each other,
    their presence on the same cache line forces the processors to frequently synchronize the cache line,
    resulting in performance penalties due to increased cache coherence traffic.

Example:

- Consider two threads updating two different variables that are located on the same cache line:
- In this example, padding1 and padding2 might end up on the same cache line,
  causing both threads to invalidate and reload the same cache line repeatedly, even though they are modifying different variables.

```java
public class FalseSharingExample {
    private static class Padding {
        public long value1;
        public long value2;
    }

    private static class Test {
        private volatile Padding padding1 = new Padding();
        private volatile Padding padding2 = new Padding();
    }

    public static void main(String[] args) {
        Test test = new Test();
        // Thread 1 modifies value1
        new Thread(() -> {
            for (int i = 0; i < 1000000000; i++) {
                test.padding1.value1++;
            }
        }).start();

        // Thread 2 modifies value2
        new Thread(() -> {
            for (int i = 0; i < 1000000000; i++) {
                test.padding2.value2++;
            }
        }).start();
    }
}
```

Mitigating False Sharing:

- Padding:
  - Introduce padding between variables to ensure that each variable resides on a different cache line.
  - pad1, pad2, ... pad7 each occupy 8 bytes, so together they occupy 56 bytes.
  - value2 is placed right after the 56 bytes of padding, starting at the beginning of a new cache line.

```
public class FixedPadding {
    public volatile long value1;
    private long pad1, pad2, pad3, pad4, pad5, pad6, pad7; // Padding fields
    public volatile long value2;
}
```

- Java @Contended Annotation:
  - Java 8 introduced the @Contended annotation (part of the java.lang.annotation package) to mitigate false sharing by ensuring that fields are placed in separate cache lines.
  - However, this requires JVM support and is not always available.

```
@Contended
public class Padded {
    public volatile long value;
}
```

- Review Data Structures:
  - Consider the layout of your data structures and their impact on cache line usage.

synchronized

Synchronized Instance Methods

The synchronization method locks all the code in the method. The lock object is this and cannot be specified manually.

A synchronized method of an object can only be accessed by one thread.

```
public synchronized void synchronisedCalculate() {
    setSum(getSum() + 1);
}
```

Synchronized Static Methods

The synchronization method locks all the code in the method. The lock object is the Class object of the current class and cannot be specified manually.

```
public static synchronized void syncStaticCalculate() {
     staticSum = staticSum + 1;
 }
```

Synchronized Blocks Within Methods

The synchronization method locks all the code in the code block. we can manually specify the lock object.

```
public void performSynchronisedTask() {
    synchronized (this) {
        setCount(getCount()+1);
    }
}

public static void performStaticSyncTask(){
    synchronized (SynchronisedBlocks.class) {
        setStaticCount(getStaticCount() + 1);
    }
}
```

Internal Implementation

The synchronized keyword uses CAS to implement a lock. When a thread acquires a lock on an object, it uses CAS to set the lock's state to locked.
If the lock is already locked, the thread will wait until the lock is released. Once the thread has acquired the lock, it can safely access the object's state.
When the thread is finished with the object, it releases the lock using CAS.
(The locking information is stored in object header.)

The monitorenter and monitorexit bytecode instructions are used to implement the synchronized keyword.
When a thread enters a synchronized block, it executes the monitorenter instruction.
This instruction attempts to acquire the lock on the monitor object.
If the lock is already held by another thread, the current thread is blocked until the lock is released.

Lock Upgrading

Biased Locking

Initial State	A lock starts in a biased state with a bias toward the first thread that acquires it.
Bias Revocation	If another thread tries to acquire the lock, the bias is revoked and the lock is upgraded.

Lightweight Locking

Lock Record	The second thread creates a lock record on its stack and attempts to acquire the lock using atomic operations.
Spin-Waiting	If the lock is not immediately available, the thread may enter a spin-wait loop, trying to acquire the lock.

Heavyweight Locking

Monitor Transition	If contention continues and multiple threads are spin-waiting, the lock is upgraded to a monitor (heavyweight lock).
Blocking	Threads that cannot acquire the lock are put into a blocked state, and the JVM manages the transition between blocked and running states using OS-level mechanisms.

Reentrancy

The synchronized keyword in Java is reentrant, meaning a thread that already holds a lock can acquire it again without any issues,
allowing nested calls to synchronized methods or blocks by the same thread.

Example:

```
public class ReentrantExample {

    public synchronized void outerMethod() {
        System.out.println("Entered outerMethod");
        innerMethod();  // Call another synchronized method
        System.out.println("Exiting outerMethod");
    }

    public synchronized void innerMethod() {
        System.out.println("Entered innerMethod");
        // Some processing
        System.out.println("Exiting innerMethod");
    }

    public static void main(String[] args) {
        ReentrantExample example = new ReentrantExample();
        example.outerMethod();
    }
}
```

Output:

```
Entered outerMethod
Entered innerMethod
Exiting innerMethod
Exiting outerMethod
```

Class Keyword

sealed

The release of Java SE 17 introduces sealed classes (JEP 409).

This feature is about enabling more fine-grained inheritance control in Java. Sealing allows classes and interfaces to define their permitted subtypes.

In other words, a class or an interface can now define which classes can implement or extend it. It is a useful feature for domain modeling and increasing the security of libraries.

Sealed Interfaces

To seal an interface, we can apply the sealed modifier to its declaration. The permits clause then specifies the classes that are permitted to implement the sealed interface:

```
public sealed interface Service permits Car, Truck {
    int getMaxServiceIntervalInMonths();
    default int getMaxDistanceBetweenServicesInKilometers() {
        return 100000;
    }
}
```

Sealed Classes

Similar to interfaces, we can seal classes by applying the same sealed modifier. The permits clause should be defined after any extends or implements clauses:

```
public abstract sealed class Vehicle permits Car, Truck {

    protected final String registrationNumber;

    public Vehicle(String registrationNumber) {
        this.registrationNumber = registrationNumber;
    }

    public String getRegistrationNumber() {
        return registrationNumber;
    }

}
```

A permitted subclass must define a modifier. It may be declared final to prevent any further extensions (the subclass should be sealed or non-sealed or final):

```
public final class Truck extends Vehicle implements Service {

    private final int loadCapacity;

    public Truck(int loadCapacity, String registrationNumber) {
        super(registrationNumber);
        this.loadCapacity = loadCapacity;
    }

    public int getLoadCapacity() {
        return loadCapacity;
    }

    @Override
    public int getMaxServiceIntervalInMonths() {
        return 18;
    }

}
```

Access Modifiers

Package-Private (default)

When no access modifier is specified, the member is package-private by default.

Inheritance:

- Same Package:
  - Package-private members can be accessed by subclasses within the same package.
- Different Package:
  - Package-private members are not accessible to subclasses in different packages.

Private

This modifier restricts access to within the class itself.

Inheritance:

- Not Accessible:
  - Private members are not accessible to subclasses.
  - They are only accessible within the class where they are defined.

Protected

This modifier allows access within the same package and by subclasses.

Inheritance:

- Same Package:
  - Protected members are accessible to subclasses within the same package.
- Different Package:
  - Protected members are accessible to subclasses in different packages.

Public

This modifier allows access from any other class.

Inheritance:

- Accessible Anywhere:
  - Public members are accessible to any class, regardless of the package or inheritance.

Definition

```
public final class Person<T, Y> extends Father<T, Y, String>  {
    // Class content
}
```

extends	Inherit from a public or protected class. A class in Java can only extend one superclass (single inheritance).
final	Indicates that this class cannot be inherited by any other class.

Constructor

A constructor has no return value and can only be used in static methods when the modifier is private.

The constructor must call the parent class constructor, and this call must be on the first line and in the subclass constructor.

```
public Person(double a,  int b, int c)  {
    super()
}
public Person(double a,  int b)  {
    super()
    this(a, b, 999)          // Call another constructor.
}

public void test(){
    System.out.println(super.name);     // Accesss the fields of the parent class.
    super.say();                        // Call the method of the parent class.
}
```

Fields

public **int a, b**=99;	normal members
public **T c**;	generic class members cannot be initialized internally, and can only receive values.
public **Car car**=new Car();	class members
public **static float money**=99.9F;	static members (it can be accessed directly by the class name: Person.money)

Methods

Member functions: reloadable, defaultable, does not support default parameters.

```
private final native void func(double a, int b, int ...va) throws SbbException, ApiException {
    // final:
    //     Prevents the method from being overridden by subclasses.
    // native:
    //     Declares a method implemented in a native language (like C/C++) using the Java Native Interface (JNI).
    //     No method body is provided in Java.
    // throws:
    //     Declares that the method may throw one or more exceptions, which the caller must handle or declare. These are
    checked exceptions.
    // throw:
    //     Used to explicitly throw an exception inside the method body.
    // Varargs (...) – Variable-Length Arguments:
    //     Internally treated as an array of the specified type (int[] in the above case).
    //     Only one varargs parameter is allowed per method, and it must be the last parameter.
    this.b += 1;
    b += 1; // Same if no local shadowing
    // Fields in the parent class are shadowed if a child declares a field with the same name. Using super allows access to the
    original.

    super.b = 5;         // Access parent field
    super.sd += 1;       // Modify inherited field
    super.func();        // Call parent class method
}
```

Inner Class or Interface

Member Inner Class

A member inner class is a non-static class that is defined within another class.

It has access to all members (including private) of the outer class.

Example:

```
  public class OuterClass {
      private String message = "Hello, World!";

      class InnerClass {
          private String message = "Hello, World!";
          void display() {
              System.out.println(message);
              System.out.println(OuterClass.this.message);
```

```
        }
    }

    public static void main(String[] args) {
        OuterClass outer = new OuterClass();
        OuterClass.InnerClass inner = outer.new InnerClass();
        inner.display();
    }
}
```

Static Nested Class

A static nested class is a static class defined within another class. It cannot access non-static members of the outer class directly.

Example:

```
public class OuterClass {
    private static String message = "Hello, World!";

    static class StaticNestedClass {
        void display() {
            System.out.println(message);
        }
    }

    public static void main(String[] args) {
        OuterClass.StaticNestedClass nested = new OuterClass.StaticNestedClass();
        nested.display();
    }
}
```

Local Inner Class

A local inner class is defined within a block, such as a method or an if statement.

It has access to the final or effectively final local variables of the block.

It has access to all members (including private) of the outer class.

Local Inner Classes in static methods cannot access instance members.

Example:

```
public class OuterClass {

    private String message = "Hello, World!";
    private static String static_message = "Hello, World!";

    void display() {
       String message = "Hello, World!";

        class LocalInnerClass {
            void printMessage() {
                System.out.println(message);
                System.out.println(OuterClass.this.message);
                System.out.println(static_message);

            }
        }

        LocalInnerClass local = new LocalInnerClass();
        local.printMessage();
    }
    static void displayStatic() {
        String message = "Hello, World!";

        class LocalInnerClass {
            void printMessage() {
                System.out.println(message);
                System.out.println(OuterClass.this.message);
                System.out.println(static_message);

            }
        }
```

```
          LocalInnerClass local = new LocalInnerClass();
          local.printMessage();
      }
      public static void main(String[] args) {
          OuterClass outer = new OuterClass();
          outer.display();
      }
  }
```

Anonymous Inner Class

An anonymous inner class is a class without a name that is used to instantiate objects with certain modifications, often used in event handling.

Example:

```
  interface Greeting {
      void greet();
  }

  public class OuterClass {
      public static void main(String[] args) {
          Greeting greeting = new Greeting() {
              @Override
              public void greet() {
                  System.out.println("Hello, World!");
              }
          };

          greeting.greet();
      }
  }
```

Abstract Class

public abstract class Father<P, S, W extends xxclass & xxinterface & xxinterface> **{**

An abstract class must contain at least one abstract method and the subclasses must override the abstract method.

A subclass can only inherit one abstract class.

int a,b=99;

public abstract void perfunc (double a, int b, int ...va) throws RuntimeException;

}

Hood Method

You can define a protected method in the superclass that calls loadProperties.

This way, you can control when loadProperties is called while also providing an option for the child classes to execute any logic needed before or after this method is called.

```
// Abstract superclass
public abstract class AbstractConfig {

    public AbstractConfig() {
        initialize();
    }

    // Hook method to allow child classes to execute additional logic
    protected void initialize() {
        loadProperties(); // Call the abstract method
    }

    // Abstract method to be implemented in child class
    protected abstract void loadProperties();
}

// Child class
public class SpecificConfig extends AbstractConfig {

    private String property;
    @Override
    protected void loadProperties() {
        // Load properties into the specific field
        this.property = "Loaded Property"; // Example implementation
    }
}
```

Freatures

Polymorphism

Polymorphism is a core concept in object-oriented programming (OOP).

It allows objects of different classes to be treated as instances of a common superclass. It supports:

- Method overriding
- Method dispatch based on runtime type

Example:

```
    Father father = new Son();  // Polymorphic reference
    father.speak();             // Executes overridden method in Son
```

Polymorphic Instantiation

Using the parent class Father to hold a child object Son:

```
Father father = new Son();       // Parent reference to child object
Father father2 = son;            // Assign Son reference to Father
```

Type Casting

Explicit (Narrowing) Type Conversion

```
    Father father = (Father) null;  // Legal but returns null
```

Downcasting:

Used when a parent reference holds a subclass object and subclass-specific methods need to be accessed.

```
    Son son = (Son) father;           // Downcasting
```

Anonymous Classes

Anonymous classes allow you to create a subclass or implement an interface on the fly.

and you can create an anonymous subclass of Father and override methods temporarily:

```
Father father = new Father() {
    int func(double a, int b, int... va) {
        return 222;
    }
};
```

Generic Class Definition

Generics in Java are compile-time only. After compilation, type parameters are erased:

```
public class Demo<T extends Comparable & Serializable> {
    private T t;

    public void add(T t) { this.t = t; }
    public T get() { return t; }
}
```

Generic Class Instantiation

```
Father<Animal> father = new Father<>();
    // Requires a type argument (e.g., Animal)
```

// Constructor can infer generic type with <>

Generic Methods

Define methods using generic types:

```
public <T extends SomeClass & Interface1 & Interface2> T func(T t) {
    return t;
}

public static <E> void printArray(E[] inputArray) {
    for (E element : inputArray) {
        System.out.printf("%s ", element);
    }
}
```

Wildcard Generics

? extends T

Accepts any type that is a subclass of T (upper bound)

? super T

Accepts any type that is a superclass of T (lower bound)

?

Unbounded wildcard (any type; treated as Object)

Example:

```
List<? extends Number> list1;  // List of Integer, Double, etc.
List<? super Double> list2;    // List of Number, Object, etc.
<F extends Enum<F>> void handleEnum(F value) { }
```

Reflection-Based Object Creation

Load class dynamically

Instantiate using no-arg or parameterized constructor

```
Class<?> clazz = Class.forName("reflection.Father");
Father father = (Father) clazz.getDeclaredConstructor().newInstance();

Constructor<?> cons = clazz.getDeclaredConstructor(String.class, String.class, int.class);
Father father2 = (Father) cons.newInstance("Li Si", "Male", 20);
```

Object Copy

Shallow Copy

A shallow copy of an object is a new instance of the same class, but the fields of the new instance reference the same objects as the original instance.

In other words, the primitive fields are copied directly, and the object references are copied, not the objects they refer to.

Deep Copy

A deep copy of an object is a new instance of the class where all fields are copied recursively.

This means that any objects referenced by the original object are also copied, not just their references.

Object Structrue

Reference:

https://shipilev.net/jvm/objects-inside-out/

Show the object structure

Object

Moving on to the actual object structure. Let us start from the very basic example of java.lang.Object. JOL would print this:

(some space will be lost, because JVM requires that the memory size occupied by a Java object should be a multiple of 8 bytes)

```
$ jdk8-64/java -jar jol-cli.jar internals java.lang.Object
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.

Instantiated the sample instance via default constructor.

java.lang.Object object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                  VALUE
      0     4        (object header)              05 00 00 00 # Mark word
      4     4        (object header)              00 00 00 00 # Mark word
      8     4        (object header)              00 10 00 00 # (not mark word)
     12     4        (loss due to the next object alignment)
Instance size: 16 bytes
```

```
Space losses: 0 bytes internal + 4 bytes external = 4 bytes total
```

Array

Arrays come with another little piece of metadata: array length. Since the object type only encodes the array element type, we need to store the array length somewhere else.

```
$ jdk8-64/bin/java -cp jol-samples.jar org.openjdk.jol.samples.JOLSample_25_ArrayAlignment
# Running 64-bit HotSpot VM.
# Using compressed oop with 3-bit shift.
# Using compressed klass with 3-bit shift.

[J object internals:
 OFFSET  SIZE   TYPE DESCRIPTION                  VALUE
      0     4        (object header)              01 00 00 00  # Mark word
      4     4        (object header)              00 00 00 00  # Mark word
      8     4        (object header)              d8 0c 00 00  # Class word
     12     4        (object header)              00 00 00 00  # Array length
     16     0   long [J.<elements>                N/A
Instance size: 16 bytes
Space losses: 0 bytes internal + 0 bytes external = 0 bytes total
```

Object Header

the object header consists of two parts: mark word and class word.

Class word

Class word carries the information about the object’s type: it links to the native structure that describes the class.

Mark word

We will talk about that part in the next section. The rest of the metadata is carried in the mark word.

There are several uses for the mark word:

Storing the metadata (forwarding and object age) for moving GCs.

Storing the identity hash code.

Storing the locking information.

Note that every single object out there has to have a mark word,

because it handles the things common to every Java object.

This is also why it takes the very first slot in the object internal structure: VM needs to access it very fast on the time-sensitive code paths, for example STW GC.

Understanding the use cases for mark word highlights the lower boundaries for the space it takes.

Contention List

Purpose

The term "Contention List" is not a standard or official term used in Java synchronization directly related to the synchronized

keyword.
However, it can be inferred to refer to the list of threads contending for access to a synchronized block or method.

Key Concepts

1) **Object Monitor**

   Every object in Java has an associated monitor that is used for synchronization.
   The monitor enforces mutual exclusion, allowing only one thread to execute the synchronized block or method at a time.

2) **Monitor States**

   - **Unlocked** No thread holds the monitor.
   - **Locked** A thread holds the monitor, and other threads are either blocked or waiting.

3) **Contention List**

   When a thread tries to enter a synchronized block or method and finds the monitor locked,
   it gets placed on a contention list. This list keeps track of all threads waiting to acquire the monitor.

4) **Wait Set**

   Threads that call Object.wait() are placed in the wait set, which is different from the contention list.

Process

1) **Initial State**
   - Monitor: Monitor is unlocked.
   - Contention List: Contention List is empty.
   - Wait Set: Wait Set is empty.
2) **Thread A Acquires the Monitor**
   - Monitor: [A] Thread A enters the synchronized block and locks the monitor.
   - Contention List: []
   - Wait Set: []
3) **Thread B Attempts to Enter the Synchronized Block**
   - Monitor: [A] Monitor is locked by Thread A.
   - Contention List: [B] Thread B is added to the contention list.
   - Wait Set: []
4) **Thread C Attempts to Enter the Synchronized Block**
   - Monitor: [A] Monitor is locked by Thread A.
   - Contention List: [B, C] Thread C is added to the contention list.
   - Wait Set: []
5) **Thread A Calls wait()**
   - Monitor: [B] Thread A releases the monitor and enters the wait set, monitor is unlocked.
     JVM grants the monitor to the next thread in the contention list (Thread B).
   - Contention List: [C]
   - Wait Set: [A]
6) **Thread B Calls notify()**
   - Monitor: [B] Monitor state remains **Locked by Thread B**.
   - Contention List: [C, A] Thread A is removed from the wait set and added back to the contention list.
   - Wait Set: []
7) **Thread B Exits the Synchronized Block**:
   - Monitor: [C] Thread B releases the monitor.
     JVM grants the monitor to the next thread in the contention list (Thread C).
   - Contention List: [A]
   - Wait Set: []

Interface

Definition

```
public interface Animal <T, Z>{
    void accept(double size);          // Abstract method to be implemented by the subclass.
    default void defaultEat(){}        // Default method can be inherited or overridden.
}
public interface Insect {
    void accept(double value);
    default void defaultEat(){}
}
public class Person implements Animal <Interger, String> {
    @Override
    public void accept(double size) {
        System.out.println("Person accepts size: " + size);
    }
}

Animal<Integer, String> animal = new Person();  // Corrected "Persion" to "Person" and specify generic types

public interface Girl <T, Z> extends Animal <T, Z> {
    int CC=77;                       // Interface fields are public, static, and final by default.
    void begin(long size);           // Interface methods are implicitly public and abstract.
             // If there is a conflict between superclass and superinterface, the method in parent class is used and no need to
             implement this method again.
    default void end() {}
             // Can be inherited, but if there is a conflict with a superclass, it overrides the superclass; if there is a conflict with
             another interface, it must be re-implemented.
    interface LittleGirl extends Girl<Double, Integer>, Insect {
      @Override
      default void accept(Double i) {        Provide different default methods for different generic types.
      }
    }
}
```

FunctionalInterface

```
@FunctionalInterface
public interface Animal {
    String apply(String b);
}

@FunctionalInterface
public interface Animal<T, R> {
             Functional interface can be represented using a lambda expression.
             T and R represent the parameter type and the return type, respectively.
             A functional interface in Java can only contain one abstract method.
                 Animal<String, String> test = (input) -> input.toUpperCase();
                 String result = test.apply("hello");
    R apply(T b);
}
```

org.springframework.web.util.HierarchicalUriComponents#UriTemplateEncoder

Enum

Basic Enum Definition

```
public enum Color {
  RED, GREEN, BLUE;
}
```

Decompilation Insight

When decompiled, enum Color looks like:

```
public final class Color extends Enum<Color> {
    public static final Color RED;
    public static final Color GREEN;
    public static final Color BLUE;
    private static final Color[] ENUM$VALUES;

    static {
        RED = new Color("RED", 0);
        GREEN = new Color("GREEN", 1);
        BLUE = new Color("BLUE", 2);
        ENUM$VALUES = new Color[] { RED, GREEN, BLUE };
           // This explains how values() returns a static cached array of enum instances.
    }
}
```

Default Methods

values()

Returns an array of all enum constants in the order they are declared.

name()

Returns the exact enum name (used in serialization)

valueOf(String str)

Returns enum constant by name (throws IllegalArgumentException if invalid)

The method is case-sensitive and expects the input string to exactly match the name of an existing enum constant, including the case.

toString()

Defaults to name(), can be overridden for custom output

Example:

```
    Color color = Color.valueOf("RED");       // OK
    System.out.println(color.name());         // RED
    System.out.println(color.toString());     // RED

    Color[] all = Color.values();             // [RED, GREEN, BLUE]
```

Enum with Fields and Methods

```
@AllArgsConstructor
@Getter
public enum Color {
    RED("R", "#FF0000"),
    GREEN("G", "#00FF00"),
    BLUE("B", "#0000FF");

    private final String code;
    private final String hex;

    @Override
    public String toString() {
        return code + ":" + hex;
    }

    @JsonCreator
    public static Color fromCode(String code) {
        for (Color c : Color.values()) {
            if (c.code.equalsIgnoreCase(code)) {
                return c;
            }
        }
        return null;
    }

    @JsonValue
    public String toJson() {
        return code;
    }
}
```

Usage:

```
Color c = Color.fromCode("G");       // GREEN
System.out.println(c.toJson());      // "G"
System.out.println(c.toString());    // G:#00FF00

Color.RED.name();       // "RED"
Color.RED.ordinal();    // 0
Color.RED.equals("RED") // false → enums are objects

Color.RED.getCode();    // "R"
Color.RED.getHex();     // "#FF0000"
```

Enum with Inner Enum and Internal Class

```
@AllArgsConstructor
@Getter
public enum StreamOpFlag {
    DISTINCT(0, set(Type.SPLITERATOR)),
    SHORT_CIRCUIT(12, set(Type.OP, Type.TERMINAL_OP));

    private final int bit;
    private final Set<Type> types;

    enum Type {
        SPLITERATOR, STREAM, OP, TERMINAL_OP, UPSTREAM_TERMINAL_OP
    }

    private static Set<Type> set(Type... values) {
        return EnumSet.copyOf(Arrays.asList(values));
    }

    private static class MaskBuilder {
        final Map<Type, Integer> map = new EnumMap<>(Type.class);

        MaskBuilder() {
            for (Type t : Type.values()) {
                map.putIfAbsent(t, 0b00);
            }
        }
        Map<Type, Integer> build() {
            return map;
        }
    }
}
```

System Result Code

```
@AllArgsConstructor
@Getter
public enum ResultCode {

    OTHER("AA", "BB", "CC", 233),      // Custom-defined fields for this constant
    SUCCESS(0, "Operation successful"), // Constructor: int code, String message
    REQ_ERROR(101, "Request error");    // Same constructor used consistently

    public final int CODE;             // Explicit value field (not auto-incremented)
    public final String MESSAGE;       // Message field for readability

    // Optional method to convert enum to a DTO or model
    public ResultCodeModel getResultCodeModel() {
        ResultCodeModel model = new ResultCodeModel();
        BeanUtils.copyProperties(this, model);
        return model;
    }
}
```

Usage

```
int code = ResultCode.REQ_ERROR.getCODE();       // 101
String message = ResultCode.SUCCESS.getMESSAGE(); // "Operation successful"
ResultCode.valueOf("OTHER ");   // ✕ Throws IllegalArgumentException (extra space)
ResultCode.valueOf("other");    // ✕ Case-sensitive – will throw an exception
ResultCode.valueOf("OTHER");    // ✅ Correct
```

Java / java.base

java.awt

BorderLayout

package java.awt;

public class **BorderLayout** implements LayoutManager2,   java.io.Serializable

CardLayout

package java.awt;

public class **CardLayout** implements LayoutManager2, Serializable

Manages components like a stack of cards, displaying only one at a time.

Useful for switching between multiple components within the same space.

Color

package java.awt;

public class **Color** implements Paint, java.io.Serializable

public Color(int r, int g, int b)

public Color(int r, int g, int b, int a)

Container

package java.awt;

public class **Container** extends Component public Component add(Component comp)

Adds a component to the container.

Dimension

package java.awt;

public class **Dimension** extends Dimension2D implements java.io.Serializable

public Dimension(int width, int height)

Defines a width and height.

Insets

package java.awt;

public class **Insets** implements Cloneable, java.io.Serializable

public Insets(int top, int left, int bottom, int right)

Defines spacing around a component.

Example: `new Insets(0, 0, AbstractLayout.DEFAULT_VGAP, AbstractLayout.DEFAULT_HGAP)`

FlowLayout

package java.awt;

public class **FlowLayout** implements LayoutManager, java.io.Serializable

Aligns components from left to right in the order they are added.

Wraps to the next row when space is insufficient.

GridLayout

package java.awt;

public class **GridLayout** implements LayoutManager, java.io.Serializable

Divides the container into an M×N grid, assigning each component to one grid cell.

GridBagLayout

package java.awt;

public class **GridBagLayout** implements LayoutManager2, java.io.Serializable

GridBagConstraints

package java.awt;

public class **GridBagConstraints** implements Cloneable, java.io.Serializable

public static final int HORIZONTAL = 2;

Robot

package java.awt;

public class **Robot**

public synchronized void mouseMove(int x, int y)

public synchronized void keyPress(int keycode)

public synchronized void keyRelease(int keycode)

public synchronized void mousePress(int buttons)

```
// Press the mouse button.
robot.mousePress(InputEvent.BUTTON1_DOWN_MASK);
// Release the mouse button.
robot.mouseRelease(InputEvent.BUTTON1_DOWN_MASK);
```

public synchronized void mouseRelease(int buttons)

java.beans

PropertyDescriptor

package java.beans;

public class **PropertyDescriptor** extends FeatureDescriptor

Describes a property in a JavaBean.

public synchronized Method getWriteMethod()

Returns the setter method.

public synchronized Method getReadMethod()

Returns the getter method.

public synchronized Class<?> getPropertyType()

Returns the property type (generic type information is lost).

Example: `class java.lang.Long`, `float`.

java.beans.FeatureDescriptor - Feature Descriptor

FeatureDescriptor

package java.beans;

public class **FeatureDescriptor**

Base class for property, method, and event descriptors.

public String getName()

Returns the name of the property, method, or event.