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.

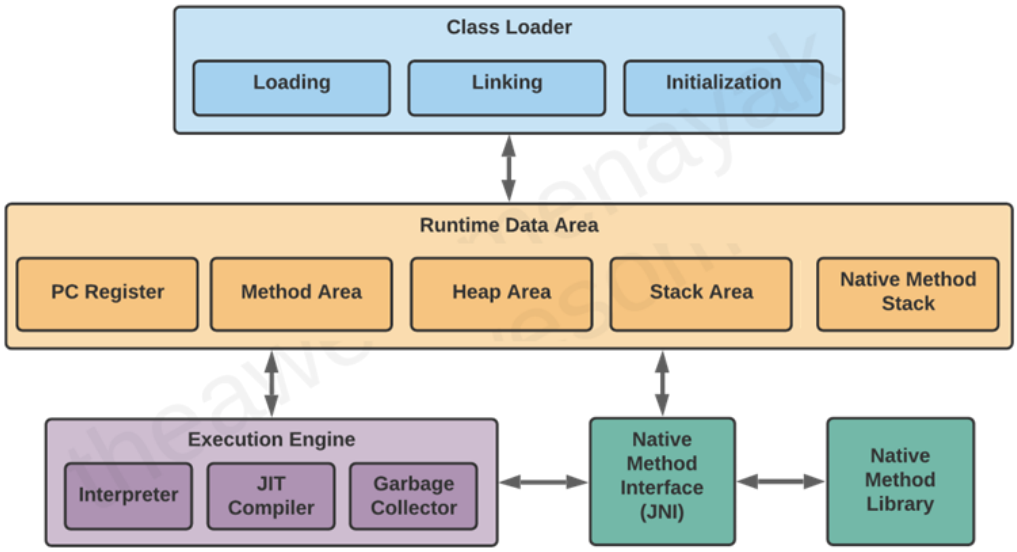
When multiple programs start, there will be multiple virtual machine instances, If the program exits or closes, the virtual machine instance will disappear.

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:
```

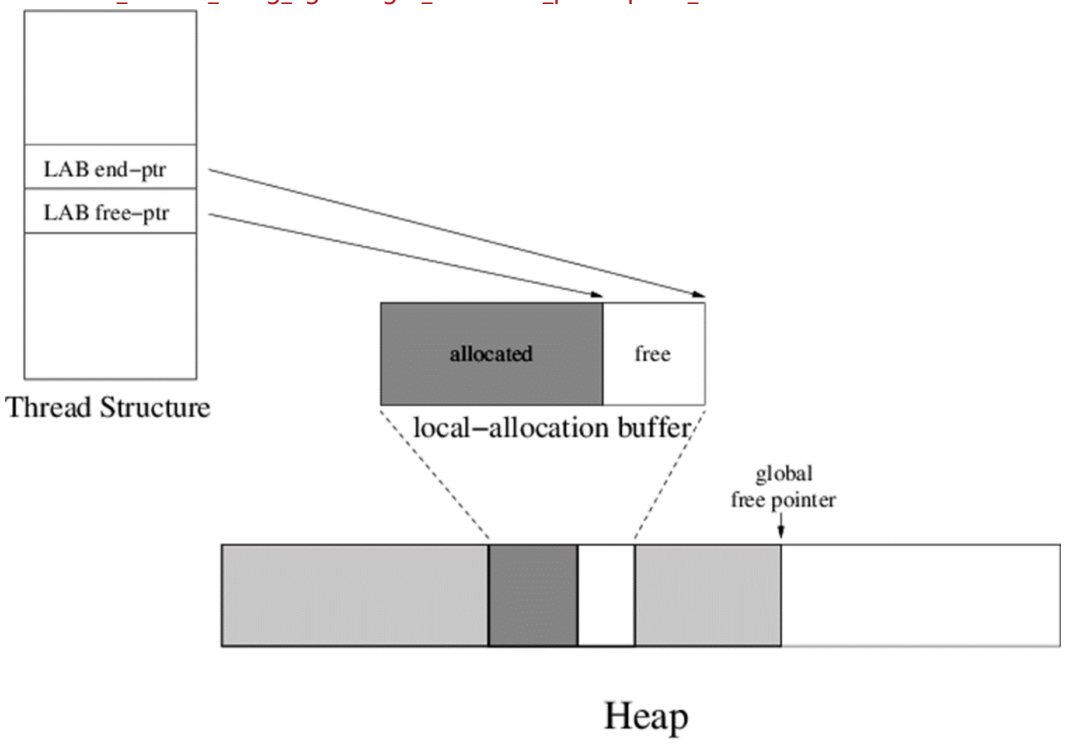
```
0: aload_0       #2    // Field w:I

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 ≤ 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
```

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
```

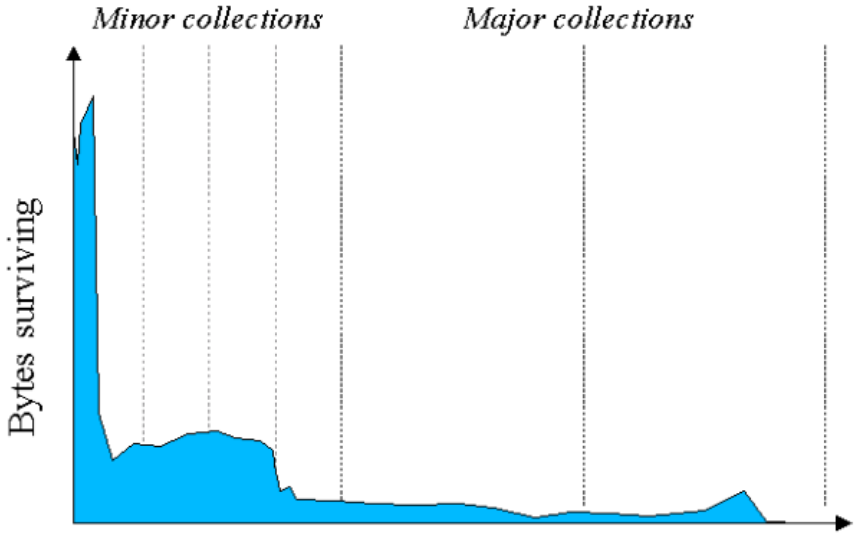
Generational Garbage Collection

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.

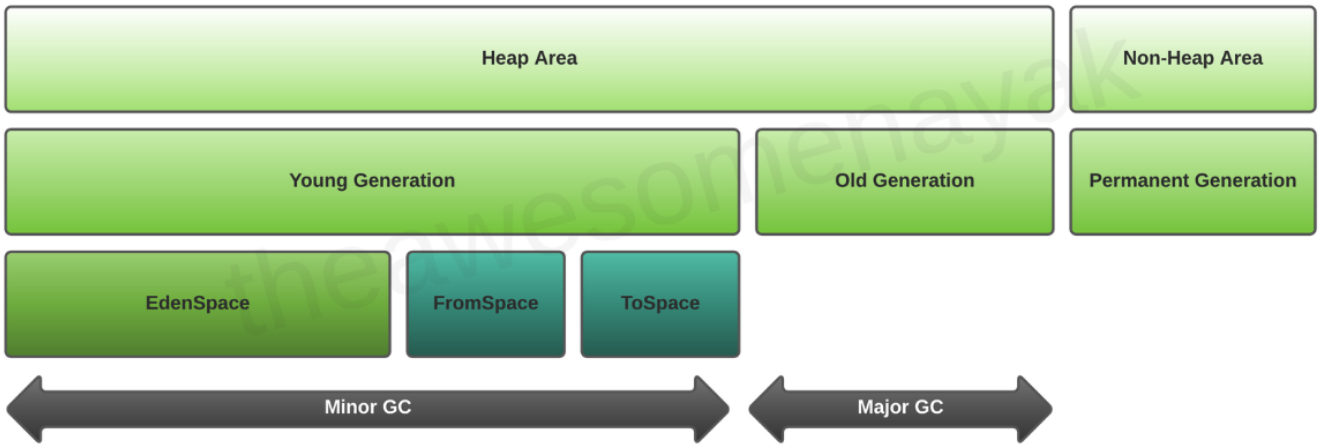
Empirical analysis of applications has shown that most objects in Java are short lived.



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

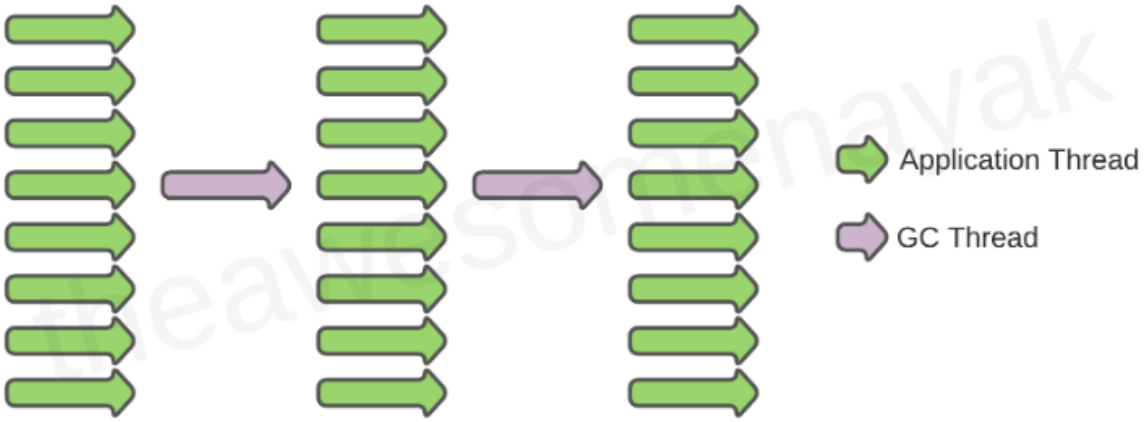
Uses a single thread for garbage collection, suitable for small applications with low memory.

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 inJDK 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

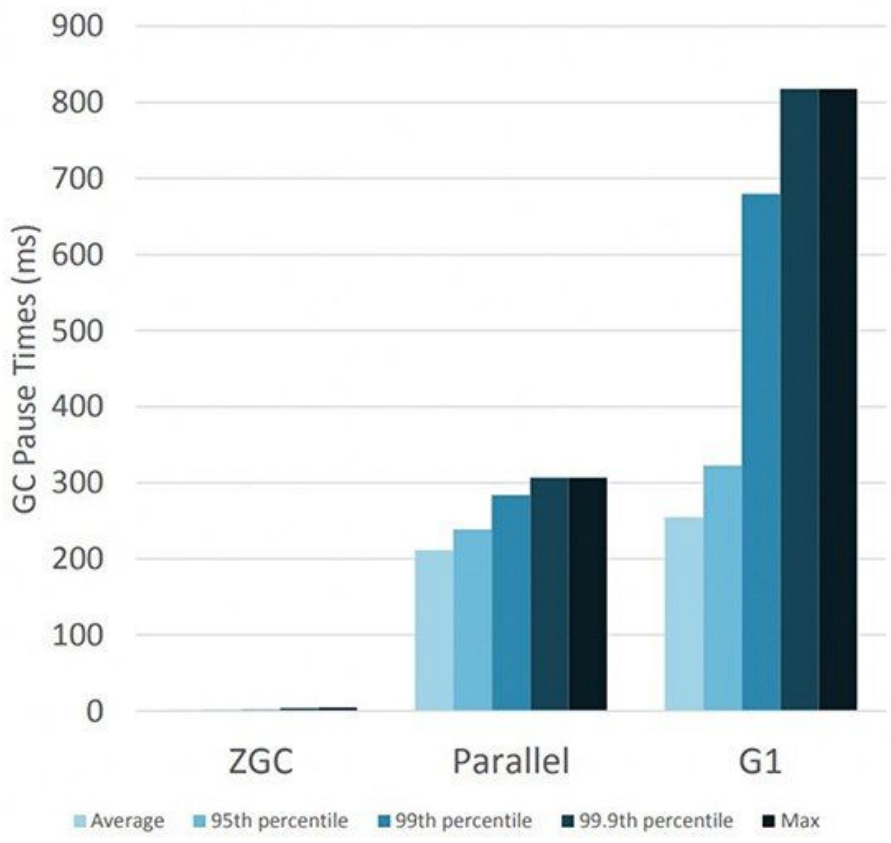
It is intended for applications which require low latency (less than 10 ms pauses) and/or use a very large heap (multi-terabytes).

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.

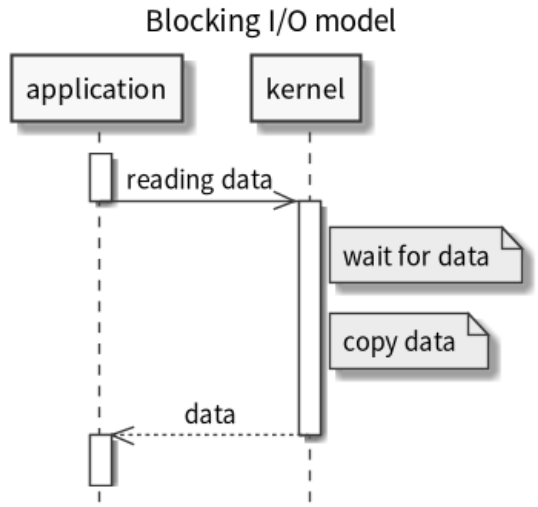
DMA (Direct Memory Access)

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).

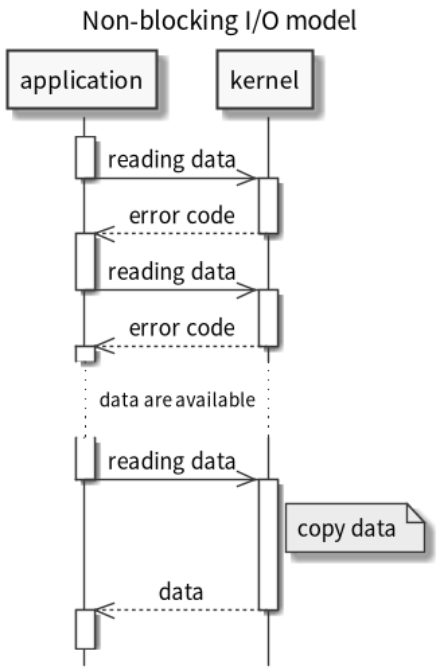
Pros

- 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.

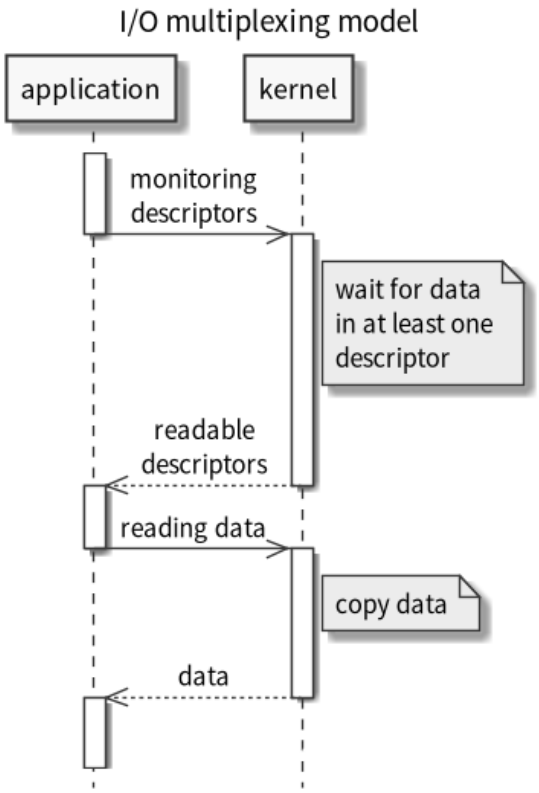
Pros:

- 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.

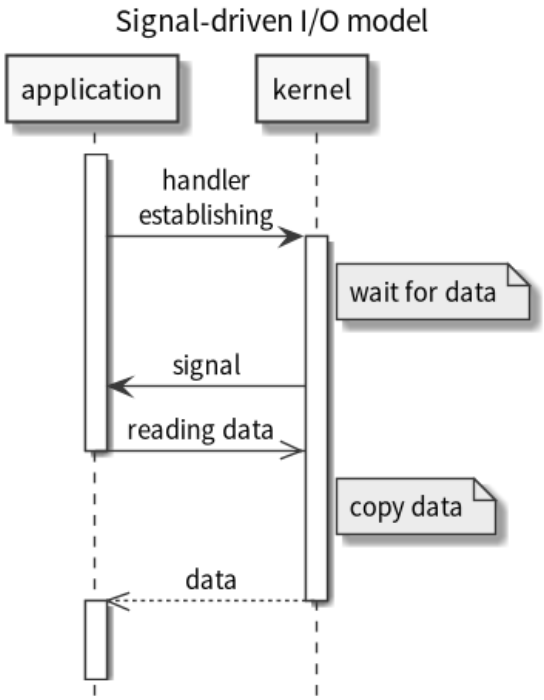
Pros:

- 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.

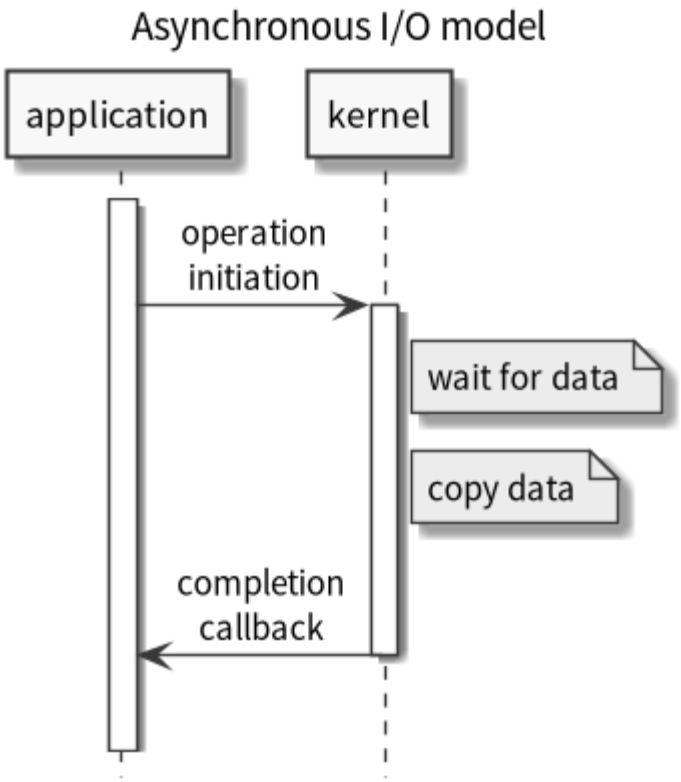
Pros:

- 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.

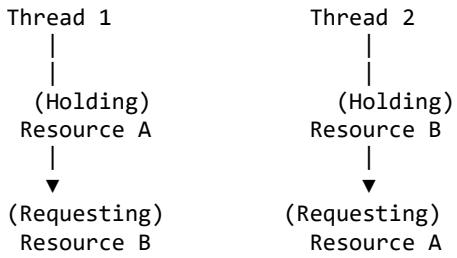
Mutual exclusion

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.



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.

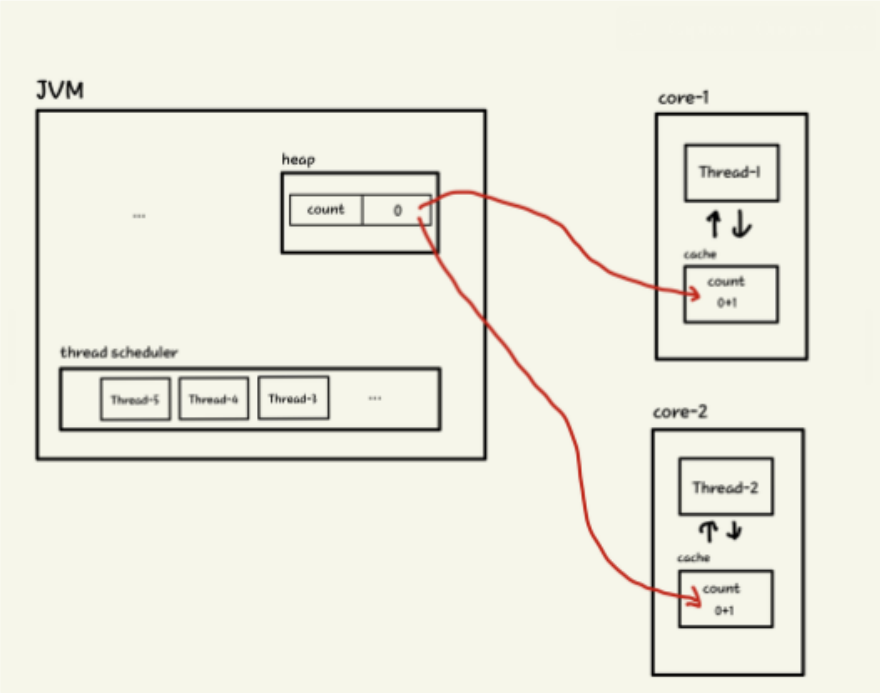
CAS

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 (Simple API for XML) is an event-driven, stream-based XML parsing method. Unlike DOM (Document Object Model), SAX doesn't load the entire XML document into memory. Instead, it reads and processes the XML document sequentially. This makes SAX a good choice for parsing large XML files efficiently.

In Java, SAX parsing involves using a DefaultHandler to handle XML events such as element start, element end, and character data. SAX is divided into two versions:

1. **SAX1**: Limited features, does not support the separation of local names, qualified names (QName), and namespace URIs.
2. **SAX2**: Improved functionality, supports local names, QNames, and URIs.

Custom Handler Implementation

```
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);
        }
```

```
  }
}
```

SAX Parser Example

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));
```

XML Example

Xml Content

```
<?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">新浪</sina:website>
   <baidu:website baidu:blog="hi.baidu.com">百度</baidu:website>
</websites>
```

sax 不支持 LocalName、QName 和 uri。对于属性 sina:blog="blog.sina.com"，sax 解析的结果是 LocalName=QName="sina:blog"，uri=""，value="blog.sina.com"。

sax2 支持 LocalName、QName、uri。对于属性 sina:blog="blog.sina.com"，sax2 解析的结果是 LocalName="blog"，QName="sina:blog"，uri=""，value="blog.sina.com"。

XPath

```
DocumentBuilder builder = DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document doc = builder.parse(new InputSource(new StringReader(xml)));
NodeList nlist = doc.getElementsByTagName("comment");

XPathFactory xpf = XPathFactory.newInstance();
XPath xp = xpf.newXPath();
NodeList nodes = (NodeList)xp.evaluate("//@*", nlist.item(0), XPathConstants.NODESET);
for(int i = 0; i < nodes.getLength(); i++)
    System.out.println(nodes.item(i));
```

Java / Core

Constants and variables

Reference:

https://www.geeksforgeeks.org/types-references-java/

Primitive Type

int a=999, **b**=a/10; **int a =1_000_000_007;** **int** a=0b11001; **int a**=0b11001; **int a**=-017; **int a**=-0xD8;

int (4 bytes, value range: -2,147,483,648 to 2,147,483,647)

The default value is 0;

short a=999,**b**=a/10; **short** a=0b11001; **short a**=0b11001; **short a**=-017; **short a**=-0xD8;

short (2 bytes)

The operation result of + is int type, and the operation result of += can be automatically converted to short.

long a=88_999L, **b**=a/10; **long** a=0b11001L; **long a**=0b11001; **long a**=-017l; **long a**=-0xD8L;

long (8 bytes)

byte a=999,**b**=a/10; **byte** a=0b11001; **byte a**=0b11001 **byte a**=-017; **byte a**=-0xD8; **byte a**='c'

byte (1 byte)

float a=3; **float a** =.75f **float a**=3.88f; **float a**=4.2F; **float a**=2e-5f; **float a**=4.4E-10F;

float, (4 bytes)

The default value is 0.0 .

double a=3; **double a**=3.88; **double a**=4.2d; **double a**=2e-5D **double a**=4.4E-10;

double, (8 bytes)

The default value is 0.0 .

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.

// 400 - 612.78 = -212.77999999999997

char a='x'; **char a**=68; **char a**='\u005d';

char (2byte)

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)

- 32 can convert uppercase letters to lowercase.

+32 can convert lowercase letters to uppercase.

Example:

```
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 a=false; **boolean b**=true;

boolean (1byte)

```
String a="xx"  String a="xx"     String a=new String("aa");
String a=new String( charArr );   String a=new String(charArr, 2, 3);
String a=new String( byteArr )   String a=new String(byteArr, 2, 3)
String a = """
content
""";
```

String

拼接效率：StringBuilder >> concat > +（字符串比较不能用==，必须 equals，因为可能出现地址和字符串的比较）

\\(在 JAVA 中双斜线才能转义，但是双斜线不能单独使用

Line break string

```
"""
 <content>
"""
```

// "a"+null+"b" = anullb

```
final int a   final void handleGet(){}   final class Person{}
```

Constant

final 常量只能被赋值一次 final 方法不能被子类方法覆盖 final 类不能被继承，并且类中所有成员 都为 final

```
int[] a;
int a[];
int a[50]{0};
int[] a={1,2,3,4};  int[] a=arr.clone();  int a[]=new int[]{1,2,3,4,5};  double[] a = new double[6]
Map<Integer, Integer>[] maps = new Map[k];
Set<Integer>[] g = new Set[5];
```

Array

不初始化默认数组内部初始值为 0，必须指定数组大小，不能超出指定大小的索引

a.clone() 返回一个克隆原始数组

a.toString() 返回首个元素，使用 Arrays.toString()返回所有元素 [33, 44, 55]

Boolean，Character，Byte，Short，Integer，Long，Float，Double 引用类型（引用的初始值为 null，基础类型初始值各不相同）

问：为什么把 if-else 写成 (c - 'a') * 2 - 1 就会快很多?

答：CPU 在遇到分支（条件跳转指令）时会预测代码要执行哪个分支，如果预测正确，CPU 就会继续按照预测的路径执行程序。但如果预测失败，CPU 就需要回滚之前的指令并加载正确的指令，以确保程序执行的正确性。

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.

Data Type Conversion

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

per instanceof Person　　当对象属于此类时 true　否则 flase　(属于子类 同时属于父类)

out

out: **for(**String val **:** arr **)**{ }　　值遍历　数组/Map/Set

assert

assert listFiles != null; 断言不为空

continue

continue out; Jump out of the specified "for" code block

break

break out; Jump out of the specified "if" code block

switch

```
cal = switch (caltype) {
   case "buddhist" -> new BuddhistCalendar(zone, aLocale);
   case "japanese" -> new JapaneseImperialCalendar(zone, aLocale);
   case "gregory"  -> new GregorianCalendar(zone, aLocale);
   default         -> null;
};
```

try catch

```
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

```
try (OutputStream out = new FileOutputStream("");OutputStream out2 = new FileOutputStream("")){
    // 只要实现的自动关闭接口(Closeable)的类都可以在 try 结构体上定义，java 会自动帮我们关闭，及时在发生异常的情况下也会。
    // ...操作流代码
} catch (Exception e) {
    throw e;
}
```

package import

package a; package a.b; 声明包

import a.b.*; import a.b.Tom; 导入包，* 代表全部的类【导入类名相同时，用包前缀指明： a.b.Tom suibian=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.

```
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 operations. The second thread creates a lock record on its stack and attempts to acquire the lock using atomic

Spin-Waiting lock. If the lock is not immediately available, the thread may enter a spin-wait loop, trying to acquire the

Heavyweight Locking

Monitor Transition (heavyweight lock). If contention continues and multiple threads are spin-waiting, the lock is upgraded to a monitor

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;
    }

}
```

Java / Features
Class
Usage

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

```
private final native void func(double a, int b, int ...va) throws SbbException, ApiException {
```

Member functions: reloadable, defaultable, does not support default parameters.

native	Java Native Interface
final	方法不可被重写
throws	方法内部可能抛出的异常，并不一定会发生这些异常。由调用者处理

throw　手动抛出一个异常。由调用者处理

子类方法重写：重写不允许新增 throw

重写返回值 必须和父类型一致，或者父类型的子类

Varargs (variable-length argument list)

The ... (ellipsis) is used to indicate that the method can accept zero or more arguments of the specified type.

The varargs parameter is internally treated as an array.

```
String[] prefixes = {"prefix1", "prefix2"};
// This is equivalent to passing those values as varargs to the method.

private static String generateARandomLineStartingWith(String ... prefixes) throws IOException {
    // prefixes is a String[] array inside the method
    for (String prefix : prefixes) {
        System.out.println(prefix);
    }
}
```

this.b+=1; b+=1; sd+=1; 可以直接修改内部成员 或者继承过来的成员（可以省略 this 访问内部成员）

super.b; **super**.sd+=1; **super**.func(); 重名覆盖会隐藏父类变量和函数，父类指针 可访问隐藏的变量和函数（可修改父类修改静态成员的值，普通成员可以修改，但不会影响到父类初始化）

throw new ApiException()

}

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 one of the core concepts of object-oriented programming (OOP) in Java.

It allows objects to be treated as instances of their parent class rather than their actual class.

This enables a single method to operate on different types of objects and perform different tasks based on the object's actual type.

Genericity

Father fa = new **Person**(){ int func (double a, int b, int ...va){return 222;} }　　匿名类创建，可以在内部临时重写方法，接收的参数 va 变成 int[]类型（适用于：函数传参，使用 new 来生成一个对象的引用，继承一个父类或抽象父类，或者实现一个接口）

Father fa =per　Father fa=(Father)null　　强制转换创建(Narrowing or Explicit Conversion)

```
List<String> lista =new ArrayList<String>( 100 ){{     初始化创建
    add("a");
    add("b");
}};
```

Father fa=new Person(); Father fa=per 多态创建，父类定义 可以接收 子类对象，在序列化过程中，接受的子类对象额外字段也会包含在其中（但是强转还是 高到低的规则）

Person per= (Person) fa 参数强转，因为父类定义可以接收子类实例，所以使用时需要强转

Person<Animal> per = new Person<>(); 泛型创建：类至少要指定一个类型，可省略构造函数的泛型 A

Class clazz=Class.forName("reflection.Person"); Person per=(Person) clazz.newInstance(); newInstane 方法创建对象

Constructor cons=clazz.getDeclaredConstructor(String.class,String.class,int.class); Person per=(Person) cons.newInstance("李四","男",20); 获取构造方法并创建对象

优化 java5 之前 object 作为动态参数，get 获取时必须强制转换的缺点 （泛型可以直接用传入的类型接收返回的类型，不指定泛型类型时，获取时还是要强制转换）

方法泛型： public <T extends SomeClassOrInterface & interface1 & interface2 & interface3> T func(T t){ } extends 可以指定继承类，也可以指定实现接口

```
public static < E > void func( E[] inputArray ){
   for ( E element : inputArray ){
      System.out.printf( "%s ", element );
   }
}
```

类泛型： public class Demo<T extends Comparable & Serializable> { T 类型就可以用 Comparable 声明的方法和 Seriablizable 所拥有的特性了

```
   private T t;
   public void add(T t) {
      this.t = t;
   }
   public T get() {
      return t;
   }
}
```

通配符泛型： <? extends T> 表示该通配符所代表的类型是 T 类型的子类（向上限制） // T extends 的语法只能在方法前的泛型声明使用

F extends Enum<F> 表示枚举类型

<? super Double> 表示该通配符所代表的类型是 Double 类型的父类（向下限制）

Wildcard types are a more powerful mechanism than object types, which can match more types, such as inheritance and implementation relationships.

类型通配符？ 类型通配符一般是使用 ? 代替具体的类型参数，单独使用可以理解成 Object，有限制会缩小范围。

类型擦除：泛型会被编译器在编译的时候去掉，生成的字节码文件中是不包含泛型中的类型信息的。// List<Object> ==> List

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

Default Method

values() 获取所有枚举值

name() 获取枚举名称，也用于序列化

valueOf(String str)　　根据获取枚举名称 获取枚举对象，不存在会报错

The method is case-sensitive and expects the input string to exactly match the name of an existing enum constant, including the case.

toString()　　获取枚举名称

Decompile

```
public enum T {
    SPRING,SUMMER,AUTUMN,WINTER;
}

public final class T extends Enum
{
    //省略部分内容
    public static final T SPRING;
    public static final T SUMMER;
    public static final T AUTUMN;
    public static final T WINTER;
    private static final T ENUM$VALUES[];    枚举类型和泛型 < 228
    static
    {
        SPRING = new T("SPRING", 0);
        SUMMER = new T("SUMMER", 1);
        AUTUMN = new T("AUTUMN", 2);
        WINTER = new T("WINTER", 3);
        ENUM$VALUES = (new T[] {
```

```
        SPRING, SUMMER, AUTUMN, WINTER
    });
    }
}
```

StreamOpFlag

```
@AllArgsConstructor
@Getter
public enum StreamOpFlag {
    DISTINCT(   0,   set(Type.SPLITERATOR).set(Type.STREAM). setAndClear(Type.OP)   ),
    SHORT_CIRCUIT(  12,    set(Type.OP). set(Type.TERMINAL_OP)   );                使用构造构建成员


    enum Type {                      内部枚举（不需要添加@Getter 和@AllArgsConstructor）
        SPLITERATOR,
        STREAM,
        OP,
        TERMINAL_OP,
        UPSTREAM_TERMINAL_OP
    }

    private static class MaskBuilder {          内部类
        final Map<Type, Integer> map;
        MaskBuilder(Map<Type, Integer> map) {
            this.map = map;
        }
        Map<Type, Integer> build() {
            for (Type t : Type.values()) {        枚举 自带获取所有值的内部函数
                map.putIfAbsent(t, 0b00);
            }
            return map;
        }
    }

}
```

ResultCode

```
@AllArgsConstructor
@Getter
public enum ResultCode {
    OTHER("AA","BB","CC", 233),       字段可自定义
    SUCCESS(0,"操作成功"),            枚举变量未赋初值，其默认的值是 0,后面的依次加 1
    REQ_ERROR(101, "请求异常");

    public final int CODE;            真实值的名称（不定义的话从 0 开始逐渐递增）
    public final String MESSAGE;      命名

   public ResultCodeModel getResultCodeModel () {
        ResultCodeModel s = new ResultCodeModel();
```

```
        BeanUtils.copyProperties(this, s);
        return s;
    }
}
```

ResultCode.PARAMETER_ERROR.CODE　　枚举对象成员值

ResultCode.valueOf("OTHER "); 根据枚举名称 获取枚举对象，而不是 private value 字段值 【找不到异常：IllegalArgumentException 】

ResultCode.values(); 返回一个装有该枚举对象 的一维数组 val.getCODE()

AccountType.REAL.equals("REAL") 永远为 false，因为枚举也是对象

Serialize

```
Enums.getIfPresent( AccountTypeEnum.class ).orNull()
@JsonCreator
public static AccountType fromValue( String text){
    for( AccountType var : AccountType.values() ){
        if( String.valueOf(var.value).equals(text) ){   // var.name().equalsIgnoreCase(text)
            return var;
        }
    }
    return null;
}

@JsonValue
public String toString(){
    return String.valueOf(value);
}
```

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.

java.io

Java built-in library offers a robust set of input/output (IO) streams that facilitate efficient data transfer between applications and external sources like files, networks, or other devices.

File

package java.io;
public class **File**

public File(String pathname)
Creates a File instance from the given pathname.
private File(String child, File parent)
Creates a File instance by automatically appending the child path to the parent.
public File(String parent, String child)
Creates a File instance by joining the parent and child paths.

public static final String separator
Path separator:
Windows: "\" (e.g., C:\Program Files\image)
Linux: "/" (e.g., /usr/local/nginx)

public static File createTempFile(String prefix, String suffix) throws IOException
Creates an empty temporary file in the default temp directory, using the specified prefix and suffix.
public boolean createNewFile()
Creates a new file in the current directory. Fails if parent directories do not exist.
public boolean mkdir()
Creates a single directory.
public boolean mkdirs()
Creates multiple directories (including parents if they do not exist).
public boolean delete()
Deletes the file or directory. Fails if the directory is not empty.

public void deleteOnExit()
    Marks the file for deletion when the JVM exits.
public long length()
    Returns the file size in bytes.
    Typically used as "(int) file.length() + 1" for BufferedReader mark parameter.
public String getAbsolutePath()
    Returns the absolute file path.
public String getParent()
    Returns the parent directory path.
public String getName()
    Returns the file or directory name.
public String getCanonicalPath() throws IOException
    Returns the absolute, normalized path (e.g., "/Users/martin6699/../test.txt" → "/Users/test.txt").

public boolean canRead()
    Checks if the file is readable.
public boolean canWrite()
    Checks if the file is writable.
public boolean exists()
    Checks if the file or directory exists.
public boolean isFile()
    Checks if it is a regular file (not a directory).
public boolean isDirectory()
    Checks if it is a directory.
public boolean isHidden()
    Checks if the file is hidden.
public long lastModified()
    Returns the last modified timestamp of the file.

public String[] list()
    Returns all filenames in the directory as a String array.
public String[] list(FilenameFilter filter)
    Returns filenames matching a filter.
public File[] listFiles()
    Returns all files and directories as File objects.
    Returns null if the caller does not exist, is a file, or lacks permissions.
public File[] listFiles(FilenameFilter filter)
    Returns all matching files and directories as File objects.

Serializable

package java.io;
public interface **Serializable**
    The serialization interface has no methods or fields and serves only to identify the semantics of being serializable.
    Serialization:
        Java serialization is a process that transforms Java objects into a sequence of bytes.
        This transformation allows us to persist objects to storage devices (like hard drives) or transmit them over networks to other applications.
        Conversely, we can convert these byte sequences back into their original Java objects, a process known as

deserialization.

Deserialization:

The process of reconstructing an object from a byte stream. This involves reading the byte stream and converting it back into an object.

serialVersionUID

A unique identifier for each serializable class, which is used during deserialization to ensure that a loaded class corresponds exactly to a serialized object.

If serialVersionUID is not declared, the JVM generates one automatically based on various aspects of the class.

Potential Issues with Automatic serialVersionUID

Inconsistency Across Compilers

Different compilers or even different versions of the same compiler may produce different serialVersionUID values for the same class if the internal implementation of the serialVersionUID computation algorithm changes.

Unintended Changes

Even minor changes to the class (like adding a method or changing a field's access modifier) can result in a different serialVersionUID.

This can break deserialization if an object serialized with an older version of the class is deserialized with a newer version.

Storage Locations for Serialized Objects

Files

Serialized objects are commonly stored in files on the filesystem. This is a straightforward way to persist the state of an object for later retrieval.

```
try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("object.ser")))
{
    oos.writeObject(myObject);
} catch (IOException e) {
    e.printStackTrace();
}
```

Memory Buffers

Objects can be serialized into memory buffers, such as ByteArrayOutputStream. This is useful for cases where you need to manipulate the serialized form in memory before further processing.

```
ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
try (ObjectOutputStream oos = new ObjectOutputStream(byteStream)) {
    oos.writeObject(myObject);
} catch (IOException e) {
    e.printStackTrace();
}
byte[] serializedBytes = byteStream.toByteArray();
```

Network Connections

Serialized objects can be transmitted over network connections by writing the byte stream to a SocketOutputStream.

This is useful for remote method invocation (RMI), distributed systems, or any client-server architecture.

```
try (Socket socket = new Socket("localhost", 12345);
     ObjectOutputStream oos = new ObjectOutputStream(socket.getOutputStream())) {
    oos.writeObject(myObject);
} catch (IOException e) {
    e.printStackTrace();
}
```

Default Serialization

When a class implements Serializable, the default serialization mechanism provided by the JVM is used.

In Java, when you use serialization, the process of saving and restoring objects is largely automatic and managed internally by the ObjectInputStream and ObjectOutputStream classes.

This mechanism serializes the object's fields automatically, except for static and transient fields.

Default Values After Deserialization for transient fields:

When an object is deserialized, transient fields are not restored to their original state. Instead, they are initialized to their default values:

For numeric types (e.g., int, float), the default value is 0.

For object references (e.g., String, custom objects), the default value is null.

Serialization Trigger Timing

The serialization process is triggered when you call the writeObject method on an ObjectOutputStream. This method is typically called by the developer to start the serialization process.

```
// Example: Serializing to a file
try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.ser"))) {
    Person person = new Person("Alice", 30);
    oos.writeObject(person); // This triggers the serialization process
} catch (IOException e) {
    e.printStackTrace();
}
```

Usage

```
import java.io.Serializable;

public class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;

    // Getters and Setters
}
```

Implement the `Serializable` Interface (Custom Serialization)

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;

public class Person implements Serializable {
    private static final long serialVersionUID = 1L;
    private String name;
    private int age;
    private transient String address; // This field will be serialized manually

    public Person(String name, int age, String address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }

    private void writeObject(ObjectOutputStream oos) throws IOException {
        // Perform the default serialization for all non-transient fields
        oos.defaultWriteObject();
        // Custom serialization for the transient field
        oos.writeUTF(address);
    }

    private void readObject(ObjectInputStream ois) throws IOException, ClassNotFoundException {
        // Perform the default deserialization for all non-transient fields
        ois.defaultReadObject();
        // Custom deserialization for the transient field
        address = ois.readUTF();
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + ", address='" + address + "'}";
    }
}
```

Serialize the Object

```
import java.io.FileOutputStream;
```

```
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializePerson {
    public static void main(String[] args) {
        Person person = new Person("Alice", 30, "123 Main St");

        try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.ser"))) {
            oos.writeObject(person);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Externalizable

package java.io;

public interface **Externalizable** extends java.io.Serializable

Only the identity of the class of an Externalizable instance is written in the serialization stream and it is the responsibility of the class to save and restore the contents of its instances.

Serializable is an automatic process, while Externalizable is a manual process.

Custom Serialization

Classes that implement Externalizable must define the writeExternal and readExternal methods to specify how the object's fields should be serialized and deserialized.

Control

Provides more control over the serialization process compared to the default serialization provided by Serializable.

void writeExternal(ObjectOutput out) throws IOException;

Specifies how to write the object's state to the `ObjectOutput`.

void readExternal(ObjectInput in) throws IOException, ClassNotFoundException;

Specifies how to read the object's state from the `ObjectInput`.

Implement the `Externalizable` Interface

```
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;

public class Person implements Externalizable {
    private String name;
    private int age;
    private transient String address; // This field won't be serialized unless explicitly handled

    // No-arg constructor required for Externalizable
    public Person() {}

    public Person(String name, int age, String address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.writeUTF(name);
        out.writeInt(age);
        out.writeUTF(address);
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
```

```
        name = in.readUTF();
        age = in.readInt();
        address = in.readUTF();
    }

    @Override
    public String toString() {
        return "Person{name='" + name + "', age=" + age + ", address='" + address + "'}";
    }
```

Serialize the Object

```
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectOutputStream;

public class SerializePerson {
   public static void main(String[] args) {
      Person person = new Person("Alice", 30, "123 Main St");

      try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.ser"))) {
         oos.writeObject(person);
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

DataInput

package java.io;

public interface **DataInput**

The DataInput interface provides for reading bytes from a binary stream and reconstructing from them data in any of the Java primitive types.

InputStream

package java.io;

public abstract class **InputStream** implements Closeable

public abstract int read() throws IOException;

Reads the next byte of data from the input stream.

Return Value:

It returns the next byte of data as an integer in the range 0 to 255. If the end of the stream is reached, it returns -1.

public int read(byte b[]) throws IOException

public int read(byte b[], int off, int len) throws IOException

public byte[] readAllBytes() throws IOException

public byte[] readNBytes(int len) throws IOException

public int readNBytes(byte[] b, int off, int len) throws IOException

public long skip(long n) throws IOException

public void skipNBytes(long n) throws IOException

public synchronized void mark(int readlimit)

public synchronized void reset() throws IOException

public int available() throws IOException
public void close() throws IOException

public boolean markSupported()
public long transferTo(OutputStream out) throws IOException

OutputStream

package java.io;
public abstract class **OutputStream** implements Closeable, Flushable
public abstract void write(int b) throws IOException;
public void write(byte b[]) throws IOException
public void write(byte b[], int off, int len) throws IOException
public void flush() throws IOException
public void close() throws IOException

(character stream)

FileReader

public class **FileReader**

public FileReader(String fileName)
public FileReader(File file)

public int read(java.nio.CharBuffer target)
public int read()
public int read(char cbuf[])
Reads characters from a file into a character array (cbuf[]) and returns the number of characters read.
public int read(char cbuf[], int off, int len)

FileWriter

public class **FileWriter**
FileWriter is a class in Java used to write character data to a file.
It is a convenience class for writing characters to files using the default character encoding of the platform.
Clears the file on opening:
When you open a file using FileWriter, it will immediately clear the file's contents unless you specify otherwise.
Creates the file if it doesn't exist:
If the file does not exist, FileWriter will create a new file.
Throws an error if the parent directory does not exist:
If the parent directory does not exist, it will throw an IOException.

public FileWriter(String fileName)
public FileWriter(String fileName, boolean append)
public FileWriter(File file)
public FileWriter(File file, boolean append)

public void write(int c)
Write a single character to the file.
Although the parameter c is of type int, it is treated as a character. Only the two low-order bytes of the integer value are written, corresponding to a single Unicode character.
public void write(char cbuf[])

public void write(char cbuf[], int off, int len)
public void write(String str)
    Writes a string to the file as-is.
public void write(String str, int off, int len)
    Writes a substring of 'str', starting from the specified offset and writing 'len' characters.
public void flush()
    Flushes the stream, ensuring all data is written to the file.
    The stream remains open for further writing.
public void close()
    Closes the stream, releasing resources.
    Automatically flushes any remaining data before closing.
    Once closed, no further writing is possible.

StringReader

package java.io;
public class **StringReader** extends Reader
    StringReader is a class in Java that provides a character stream whose source is a string.
    It extends the Reader class and allows reading characters from a String as if it were a stream.
    This is useful for parsing or processing text data stored in a string format.

public StringReader(String s)

public int read() throws IOException
public int read(char cbuf[], int off, int len) throws IOException

StringWriter

public class **StringWriter** extends Writer

BufferedReader

package java.io;
public class **BufferedReader** extends Reader
        The BufferedReader class provides buffering capabilities to improve the efficiency of reading character data from a stream.

private static int defaultCharBufferSize = 8192;

public BufferedReader(Reader in, int sz)
    Creates a new buffered character input stream that uses a specified size for the input buffer.
public BufferedReader(Reader in)
    Creates a new buffered character input stream with a default-sized input buffer.

String readLine(boolean ignoreLF)
public void mark(int readAheadLimit)
        Mark the current position in the input stream so that the stream can be "reset" back to this point later.
        Character Limit:
            This limit defines how many characters can be read after calling mark before the mark becomes invalid.
            If you read more than this limit, the reset() operation may fail because the mark might not be preserved anymore.
        Buffer Size Impact:
            If the readAheadLimit is larger than the default input buffer size, a new buffer will be allocated with a size at least equal to the readAheadLimit.

This could impact memory usage, so it's essential to use larger values cautiously to avoid excessive memory allocation.

public void reset() throws IOException

public boolean ready() throws IOException

Usage

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class BufferedReaderExample {
    public static void main(String[] args) {
        try (BufferedReader br = new BufferedReader(new FileReader("example.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                System.out.println(line); // Process each line of the file
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

BufferedWriter

package java.io;

public class **BufferedWriter** extends Writer

Character Buffered Output Stream

public BufferedWriter(Writer out, int sz)

Creates a new buffered character output stream with the specified buffer size

public BufferedWriter(Writer out)

Creates a new buffered character output stream with the default buffer size

public void write(int c)

public void write(char cbuf[], int off, int len)

public void write(String s, int off, int len)

Writes the string `s` to the file in append mode (can be used in a loop to write parts of an array)

public void newLine()

Writes a line separator, the separator string is defined by the system property

public void flush()

public void close()

PrintWriter

package java.io;

public class **PrintWriter** extends Writer

PrintWriter in Java is a convenience class for writing formatted text to a character-output stream.

Usage1

```
public class PrintWriterExample {
    public static void main(String[] args) {
        try (PrintWriter writer = new PrintWriter("output.txt")) {
            writer.println("Hello, World!");
            writer.println(123);
            writer.printf("Formatted number: %.2f%n", 456.789);
        } catch (Exception e) {
            e.printStackTrace();
```

```
        }
    }
}
```

Usage2

```
import java.io.PrintWriter;
import java.io.File;

public class CSVWriterExample {
    public static void main(String[] args) {
        File csvFile = new File("data.csv");

        try (PrintWriter writer = new PrintWriter(csvFile)) {
            // Writing header row
            writer.println("Name,Age,City");

            // Writing entire rows directly
            writer.println("Alice,30,New York");
            writer.println("Bob,25,San Francisco");
            writer.println("Charlie,28,Los Angeles");

            System.out.println("CSV file created successfully: " + csvFile.getAbsolutePath());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

(byte stream)

FileInputStream

package java.io;

public class FileInputStream　字节输入流（音频视频图片）

因为字节流一次读一个字节,而不管 GBK 还是 UTF-8 一个中文都是多个字节,用字节流每次只能读其中的一部分,所以就会出现乱码问题。

打开大文件

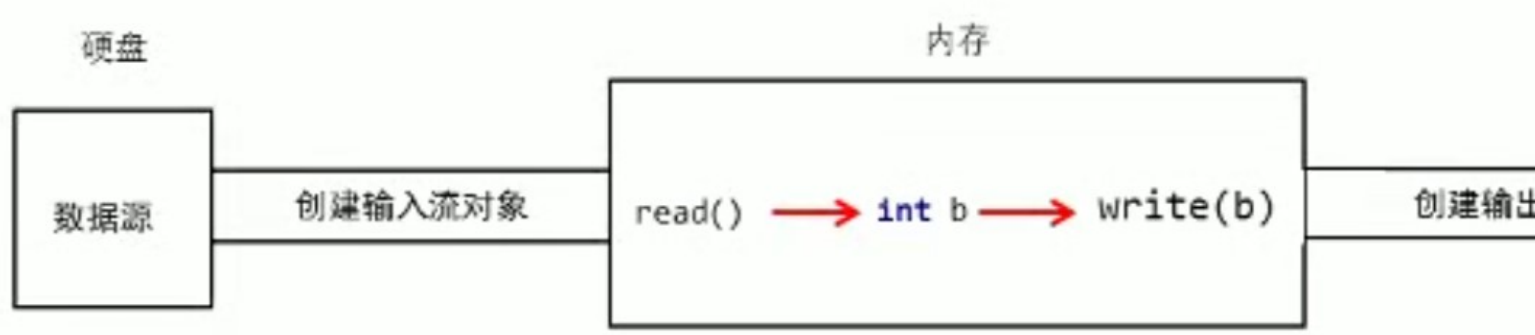
打开大文件的关键在于，不能直接将文件中的数据全部读取到内存中，以免引发 OOM。要使用较小的内存空间来解决问题。（每次读取文件中的一部分内容，分多次处理这个文件）

如果我们打开的是文本文件，期望读取甚至分析该文件中的内容，则可以采用 java.util.Scanner 来逐行读取文件的内容。在 Scanner 遍历文件的过程中，每处理一行之后，我们都要丢弃对该行的引用，以节约内存。

如果我们打开的是字节文件，期望拷贝或者搬运该文件中的内容，则可以采用缓冲流或 NIO。每次利用缓冲区处理文件中的一小段数据，这样在处理过程中使用的内存空间便是很有限的，不会造成内存溢出的问题。

public FileInputStream(String name)

public FileInputStream(File file)



public int read()　　读取 下一个字节/字符

public int read(byte b[])　　读取字节/字符到数组中（返回的是读入缓冲区的总字节数,也就是实际的读取字节个数）
public int read(byte b[], int off, int len)　读取字节/字符到数组中从索引 off 存到 len　（接近末尾数组可能不读满，末尾返回-1）
public void close() throws IOException　关闭流（打开就要关闭）

FileOutputStream

public class FileOutputStream

public FileOutputStream(String name)
public FileOutputStream(File file)
public FileOutputStream(String name, boolean append)　　创建文件输出流，如果第二个参数为 true，追加内容

public void write(int b)　　一次写一个字节数据（实际上写到文件中的,是这个整数在码表中对应的那个字符）
public void write(byte b[])　　一次写一个字节数组数据
public void write(byte b[], int off, int len)　一次写一个字节数组的部分数据
public void flush()　写入后清空流（强制将所有缓冲的输出字节被写出）
public void close()　关闭流（使用 finally 保证一定执行）

ByteArrayInputStream

package java.io;
public class ByteArrayInputStream extends InputStream

ByteArrayOutputStream

package java.io;
public class **ByteArrayOutputStream** extends OutputStream

No Need to Close:

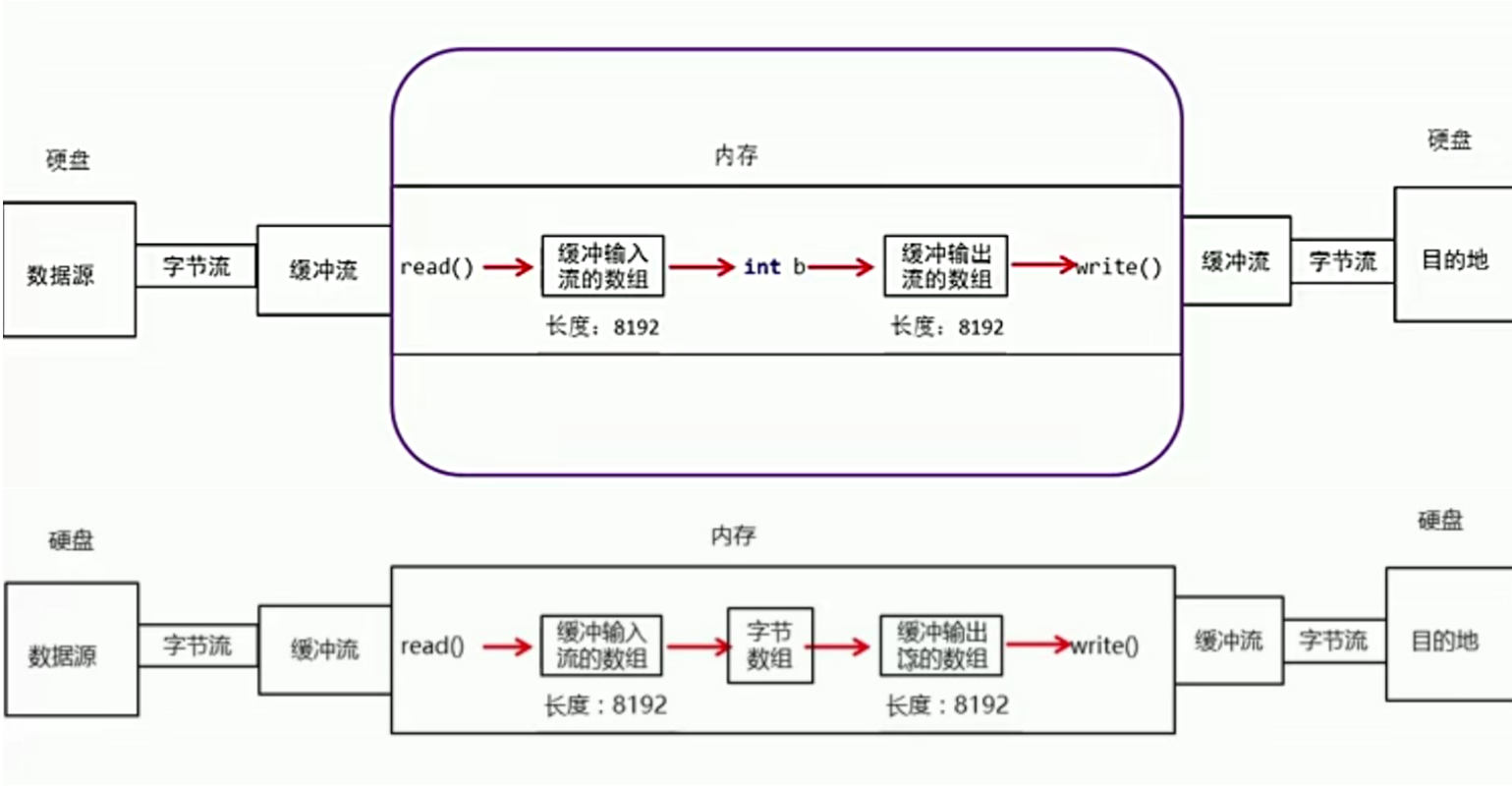
Unlike many other I/O streams in Java, you don't need to close a ByteArrayOutputStream.
The close() method has no effect, and the stream can still be used after being "closed".
This is because ByteArrayOutputStream does not hold any resources like file handles or network sockets that require releasing.

Usage

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
baos.write(65); // Writes the byte 'A' (ASCII value 65)
baos.write("Hello, World!".getBytes()); // Writes a byte array to the stream

byte[] result = baos.toByteArray(); // Retrieves the accumulated data as a byte array
System.out.println(new String(result)); // Prints "AHello, World!"
```

BufferedInputStream

package java.io;
public class **BufferedInputStream** extends FilterInputStream



protected int markpos = -1; The value of the pos field at the time the last mark method was called.
protected int pos; The current position in the buffer.

public BufferedInputStream(InputStream in)
public BufferedInputStream(InputStream in, int size)

public synchronized int read() throws IOException
public synchronized int read(byte b[], int off, int len) throws IOException
private void fill() throws IOException Delegates basic components for deeper operations.

Override
public int read(byte b[]) throws IOException

Reads up to b.length bytes of data from this input stream into an array of bytes. This method blocks until some input is available.

Params:

b – the buffer into which the data is read.

Returns:

the total number of bytes read into the buffer, or -1 if there is no more data because the end of the stream has been reached.

Throws:

IOException – if an I/O error occurs.

See Also:

read(byte[], int, int)

BufferedOutputStream

package java.io;
public class **BufferedOutputStream** extends FilterOutputStream 字节缓冲输出流

public BufferedOutputStream(OutputStream out)

public synchronized void write(int b)
public synchronized void write(byte b[], int off, int len)

PushbackInputStream

package java.io;
public class **PushbackInputStream** extends FilterInputStream

A PushbackInputStream adds functionality to another input stream, namely the ability to "push back" or "unread" byptes by storing pushed-back byptes in an internal buffer.
The PushbackInputStream class provides the ability to "push back" or "unread" bytes to the input stream.
This feature is useful when parsing data, allowing you to read ahead and then put back bytes if you decide they should not have been read.

public PushbackInputStream(InputStream in, int size)
public PushbackInputStream(InputStream in)

public void unread(byte[] b) throws IOException

Pushes back an array of bytes by copying it to the front of the pushback buffer.
After this method returns, the next byte to be read will have the value b[0], the byte after that will have the value b[1], and so forth.
Params:
b – the byte array to push back
Throws:
NullPointerException – If b is null.
IOException – If there is not enough room in the pushback buffer for the specified number of bytes, or this input stream has been closed by invoking its close() method.

public void unread(int b) throws IOException

Pushes back a byte by copying it to the front of the pushback buffer. After this method returns, the next byte to be read will have the value (byte)b.
Params:
b – the int value whose low-order byte is to be pushed back.
Throws:
IOException – If there is not enough room in the pushback buffer for the byte, or this input stream has been closed by invoking its close() method.

Usage

```
import java.io.ByteArrayInputStream;
import java.io.IOException;
import java.io.PushbackInputStream;

public class PushbackInputStreamExample {
    public static void main(String[] args) {
        String inputString = "HelloABCWorld";
        byte[] inputBytes = inputString.getBytes();

        // Create a ByteArrayInputStream from the input bytes
        ByteArrayInputStream byteArrayInputStream = new ByteArrayInputStream(inputBytes);

        // Create a PushbackInputStream with a pushback buffer of the same size as the input
        try (PushbackInputStream pushbackInputStream = new PushbackInputStream(byteArrayInputStream,
inputBytes.length)) {
```

```
            byte[] buffer = new byte[3];
            int bytesRead;

            while ((bytesRead = pushbackInputStream.read(buffer)) != -1) {
                // Check if we have read "ABC"
                String readString = new String(buffer, 0, bytesRead);
                if (readString.equals("ABC")) {
                    // If "ABC" is found, push it back
                    pushbackInputStream.unread("ABC".getBytes());

                    // Replace "ABC" with "XYZ"
                    byte[] replacement = "XYZ".getBytes();
                    for (byte b : replacement) {
                        System.out.print((char) b);
                    }

                    // Skip over the next three bytes ("ABC")
                    pushbackInputStream.skip(3);
                } else {
                    // Otherwise, print the read bytes
                    for (int i = 0; i < bytesRead; i++) {
                        System.out.print((char) buffer[i]);
                    }
                }

                // Reset the buffer
                buffer = new byte[3];
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Usage

```
import java.io.*;

public class PushbackInputStreamExample {
    public static void main(String[] args) throws IOException {
        String str = "Hello, World!";
        byte[] data = str.getBytes();

        // Wrap a PushbackInputStream around a ByteArrayInputStream
        try (PushbackInputStream pbis = new PushbackInputStream(new ByteArrayInputStream(data))) {
            int ch;
            while ((ch = pbis.read()) != -1) {
                if (ch == ',') {
                    // Push back the comma and break
                    pbis.unread(ch);
                    break;
                }
                System.out.print((char) ch); // Print each character until the comma
            }

            // After pushing back, read again
            ch = pbis.read();
            System.out.println("\nPushed back and read again: " + (char) ch); // This will print the comma
        }
    }
}
```

DataInputStream

package java.io;

public class **DataInputStream** extends FilterInputStream implements DataInput

DataInputStream is a class in Java that allows an application to read primitive Java data types (e.g., int, float, long, etc.)

from an underlying input stream in a machine-independent way.

It extends the FilterInputStream class, which means it acts as a wrapper around another InputStream, providing additional functionality for reading data in a binary format.

public final int read(byte b[]) throws IOException

public final int read(byte b[], int off, int len) throws IOException

PrintStream

package java.io;

public class **PrintStream** extends FilterOutputStream implements Appendable, Closeable

The PrintStream class provides methods to write data to an output stream with additional functionality, such as formatting and automatic flushing.

Usage

```
import java.io.PrintStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class PrintStreamExample {
    public static void main(String[] args) {
        try (PrintStream ps = new PrintStream(new FileOutputStream("output.txt"))) {
            ps.println("Hello, World!"); // Writes a string followed by a newline
            ps.printf("Number: %d, Float: %.2f%n", 42, 3.14); // Formatted output
            ps.append("Appending text"); // Appends text
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

FilterInputStream

package java.io;

public class **FilterInputStream** extends InputStream

FilterInputStream is designed to provide a way to wrap another InputStream and add additional functionality or modify the data as it is read.

You can override the read methods to provide custom behavior

protected volatile InputStream in;

protected FilterInputStream(InputStream in)

public int read(byte b[]) throws IOException

Usage

```
import java.io.FilterInputStream;
import java.io.InputStream;
import java.io.IOException;

public class CustomFilterInputStream extends FilterInputStream {

    public CustomFilterInputStream(InputStream in) {
        super(in);
    }

    @Override
    public int read() throws IOException {
        int data = super.read();
        // Add custom behavior, e.g., modifying the data
        if (data != -1) {
            // Example: Convert to uppercase (if reading text data)
            data = Character.toUpperCase(data);
        }
```

```
        return data;
    }

    @Override
    public int read(byte[] b, int off, int len) throws IOException {
        int bytesRead = super.read(b, off, len);
        // Add custom behavior here if needed
        return bytesRead;
    }
}
```

FilterOutputStream

package java.io;

public class **FilterOutputStream** extends OutputStream

ObjectInputStream

public class **ObjectInputStream**

public ObjectInputStream(InputStream in)

public final Object readObject()

ObjectOutputStream

public class ObjectOutputStream

public ObjectOutputStream(OutputStream out)

public final void writeObject(Object obj)

(conversion stream)

InputStreamReader

public class **InputStreamReader**

Converts the byte sequence read from the input stream into characters based on the specified character encoding.

public String getEncoding()

Usage

```
import java.io.*;

public class CharacterStreamExample {
    public static void main(String[] args) throws IOException {
        // Reading from a file
        InputStreamReader reader = new InputStreamReader(new FileInputStream("input.txt"), "UTF-8");
        int c;
        while ((c = reader.read()) != -1) {
            System.out.print((char) c);
        }
        reader.close();

        // Writing to a file
        OutputStreamWriter writer = new OutputStreamWriter(new FileOutputStream("output.txt"), "UTF-8");
        writer.write("Hello, world!");
        writer.close();
    }
}
```

OutputStreamWriter

public class OutputStreamWriter

Converts the character sequence to a byte sequence based on the specified character encoding before writing it to the output stream.

public OutputStreamWriter(OutputStream out, String charsetName)
public OutputStreamWriter(OutputStream out)
public OutputStreamWriter(OutputStream out, Charset cs)
public OutputStreamWriter(OutputStream out, CharsetEncoder enc)

public void write(int c)
public void write(char cbuf[], int off, int len)
public void flush()
public void close()

(security)

FilePermission

package java.io;
public final class **FilePermission** extends Permission implements Serializable　文件权限，权限的顶级类的子类

(annotation)

@Serial　检查序列化成员（字段和方法），如果它与配置文件不匹配，编译器应该提醒您一个错误。并且它向读者提供了类似的提示，该成员将被序列化使用。

java.lang

(Core)

AutoCloseable

public interface **AutoCloseable**　自动关闭资源
void close() throws Exception;　关闭系统资源

Class

package java.lang;
public final class **Class**<T>　代理类，任何类被使用时，系统都会为之建立一个 java.lang.Class 对象 （类名.class==代理类对象）

public static Class<?> forName(String className) throws ClassNotFoundException　返回一个类，要求 JVM 查找并加载指定的类
public static Class<?> forName(String name, boolean initialize, ClassLoader loader) throws ClassNotFoundException　使用给定的类加载器返回与具有给定字符串名称的类或接口关联的 类对象。
public String getSimpleName()　获取类名
public String getName()　获取全类名（内部类显示不同：zengqiang.Log4jTest$Innr）
public String getCanonicalName()　获取全类名　// zengqiang.Log4jTest.Innr

public Field[] getFields() throws SecurityException　返回所有 公共成员变量 对象数组
public Field[] getDeclaredFields() throws SecurityException　返回所有 成员变量 对象数组
public Field getField(String name) throws NoSuchFieldException, SecurityException　返回单个 公共成员变量 对象
public Field getDeclaredField(String name) throws NoSuchFieldException, SecurityException　返回单个 成员变量 对象

public Method[] getMethods() throws SecurityException　返回所有 公共成员方法 对象数组

public Method[] getDeclaredMethods() throws SecurityException　　返回所有 成员方法 对象数组
public Method getMethod(String var1, Class... var2) throws NoSuchMethodException, SecurityException　　返回一个和参数配型相符的 公共成员方法 对象
public Method getDeclaredMethod(String var1, Class... var2) throws NoSuchMethodException, SecurityException　　返回一个和参数配型相符的 成员方法 对象

public Constructor<?>[] getConstructors() throws SecurityException　　返回所有 公共构造方法 对象数组
public Constructor<?>[] getDeclaredConstructors() throws SecurityException　　返回所有 构造方法 对象数组
public Constructor<T> getConstructor(Class... parameterTypes) throws NoSuchMethodException, SecurityException　　返回一个和参数配型相符的 公共构造方法 对象
public Constructor<T> getDeclaredConstructor(Class... parameterTypes) throws NoSuchMethodException, SecurityException　　返回一个和参数配型相符的 构造方法 对象　　// clazz.getDeclaredConstructor(String.class,String.class,int.class)
public Class<?>[] getInterfaces()　　获取这个类或接口　直接实现的所有接口

public T newInstance() throws InstantiationException, IllegalAccessException　　返回当前类的实例对象，只能调用无参构造（如果是非 public 的，需要使用 setAccessible 临时取消检查，然后再创建对象）

public <A extends Annotation> A getAnnotation(Class<A> annotationClass)　　获取在当前类上使用的 指定注解类型
public boolean isAnnotationPresent(Class<? extends Annotation> annotationClass)　　当前类上面是否有 指定注解类型
native Class<? super T> getSuperclass()　　获取直接继承的父类类型（不包含泛型参数）
　　如果此 Class 表示 Object 类、接口、基本类型或 void，则返回 null
　　如果此对象表示一个数组类，则返回表示该 Object 类的 Class 对象
Type getGenericSuperclass()　　获取直接继承的父类类型
　　如果此 Class 表示 Object 类、接口、基本类型或 void，则返回 null
　　泛型的类型是无法在运行时通过反射取得的，泛型类型在编译成字节码的时候已经被虚拟机给去掉了，只是起到提示编译器进行类型检查的作用
　　包含泛型参数，返回的 Type instanceof ParameterizedType 时，可以强转为 ParameterizedType 获取泛型
native boolean isAssignableFrom(Class<?> cls)　　是否是某个类的父类（相比于 instanceof，为 class 判断）
T[] getEnumConstants()　　获取枚举值数组
native boolean isInterface()　　是否是接口

public java.net.URL getResource(String name)　　获取资源文件绝对路径（不能获取内部目录路径，但是能获取根目录路径"/"）
　When working with file paths from resources in a JAR, getPath() may return a path that starts with a leading /,
　which is not a valid path on some operating systems (especially on Windows).
　Illegal char <:> at index 2: /C:/Users/simi/Desktop/DevProjects/simi/simi-app/simi-review-tool/target/classes/storage.txt
　Solution:
```
Path.of(REVIEW_LOG_FILE_PATH.toURI()).toFile()
```

public InputStream getResourceAsStream(String name)　　获取资源文件的流
　// getClass().getResourceAsStream("/V1.2_price.xlsx")　　读取 resources 路径下的 V1.2_price.xlsx
public T cast(Object obj)　　类型转换，可以直接使用赋值强转　　// Father fa = Father.class.cast(per)
public native Class<?> getComponentType()　　返回一个数组成员类型，如果不是数组，返回 null　　// T[] array1 array1.getClass().getComponentType()

1. 通过 invoke 调用实例的方法

   obj = Class.forName("org.chen.yuan.reflect.Person");

   Method met = c1.getMethod("sayChina");

   met.invoke(obj.newInstance());

2. Access some fields in internal JVM classes and print out the number of object references:

```
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);
```

Object

package java.lang;

public class **Object**

public final native Class<?> getClass();

public String toString()

public native int hashCode()

- Consistency:
  - If the equals method of an object is not overridden, the default hashCode method returns an integer that is consistent within a single execution of the application.
  - This means the hash code will not change while the object remains in memory.
- Default Hash Code Consistency:
  - Mutable Objects
    - For mutable objects, if the hashCode method depends on the object's fields, changing those fields can affect the object's hash code.
    - This means that if you modify the fields used to compute the hash code, the hash code value will change as well.
  - Immutable Objects
    - For immutable objects, the hash code value is computed once and remains constant throughout the object's lifetime.
    - This means that any changes to the fields of the object do not affect its hash code because the fields themselves cannot be changed after the object is created.

```
String str1 = "Hello";
String str2 = "Hello";

System.out.println(str1.hashCode()); // Same hash code for both strings
System.out.println(str2.hashCode()); // Same hash code for both strings
```

- Size
  - The hashCode() method of the Object class returns a 32-bit integer (4 bytes). The value of this integer can be any valid int value within the range of -2^31 to 2^31 - 1.
  - Default hash code example: 987654321
- Reasons for Overriding hashCode When Overriding equals
  - Maintaining Contract Consistency:
    - The general contract of hashCode is that if two objects are considered equal according to the equals method, they must return the same hash code.

If equals is overridden but hashCode is not, objects that are equal might produce different hash codes, which can lead to unexpected behavior in hash-based collections.

Correctness in Hash-Based Collections:

Hash-based collections like HashMap use the hash code to organize and locate objects efficiently.

When you override equals, hash-based collections rely on hashCode to ensure that equal objects are placed in the same bucket.

If hashCode is not overridden properly, the collection may behave incorrectly, such as failing to locate objects or having duplicate entries.

public final void wait() throws InterruptedException

Causes the current thread to wait until another thread invokes notify() or notifyAll() on the same object.

The wait method causes the current thread to release the lock on its synchronization object (also called a monitor), enter the wait queue and be blocked.

After being notified, the thread must reacquire the monitor before it can proceed.

wait() is more appropriate when threads need to wait for specific conditions to be met

public final native void notify();

Wakes up a single thread that is waiting on the object's monitor.

public final native void notifyAll();

Wakes up all threads that are waiting on the object's monitor.

protected void finalize() throws Throwable

Called by the garbage collector on an object when garbage collection determines that there are no more references to the object.

A subclass overrides the finalize method to dispose of system resources or to perform other cleanup.

GC calls this method before reclaiming the current object.

```
public boolean equals(Object obj)
```

If an object does not override the equals method, the default implementation inherited from the Object class is used.

This default implementation compares object references, not the actual contents of the objects.

```
{
    return (this == obj);
}
```

Producer-Consumer Scenario

```
import java.util.LinkedList;
import java.util.Queue;

public class ProducerConsumer {
    private final Queue<Integer> queue = new LinkedList<>();
    private final int CAPACITY = 5;
    private final Object lock = new Object();

    public void produce() throws InterruptedException {
        int value = 0;
        while (true) {
            synchronized (lock) {
                while (queue.size() == CAPACITY) {
                    lock.wait(); // Wait until there is space in the queue
                }
                queue.offer(value++);
                System.out.println("Produced: " + value);
                lock.notify(); // Notify any waiting consumer
            }
            Thread.sleep(1000); // Simulate production time
```

```
        }
    }

    public void consume() throws InterruptedException {
        while (true) {
            synchronized (lock) {
                while (queue.isEmpty()) {
                    lock.wait(); // Wait until there is an item to consume
                }
                int value = queue.poll();
                System.out.println("Consumed: " + value);
                lock.notify(); // Notify any waiting producer
            }
            Thread.sleep(1000); // Simulate consumption time
        }
    }

    public static void main(String[] args) {
        ProducerConsumer pc = new ProducerConsumer();

        Thread producerThread = new Thread(() -> {
            try {
                pc.produce();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        Thread consumerThread = new Thread(() -> {
            try {
                pc.consume();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        producerThread.start();
        consumerThread.start();
    }
}
```

Runtime

package java.lang;

public class **Runtime**

public static Runtime getRuntime()　　获取当前运行环境

public native long freeMemory()　　虚拟机可用内存

public native int availableProcessors();　虚拟机可用处理器

(Data Operation)

Math

package java.lang;

public final class **Math**

static int max(int a, int b)

Returns the maximum of two integers.

static int min(int a, int b)

Returns the minimum of two integers.

static float abs(float a)

Returns the absolute value of a float value.

static double abs(double a)

Returns the absolute value of a double value.

static double random()
    Returns a random value in the range [0.0, 1.0).
static double floor(double a)
    Returns the largest integer less than or equal to the argument.
static double ceil(double a)
    Returns the smallest integer greater than or equal to the argument.
static double rint(double a)
  Returns the closest integer to the argument, with ties (.5) rounded to the nearest even integer.
static int round(float a)
    Rounds a float value to the nearest integer (returns int).
static long round(double a)
    Rounds a double value to the nearest integer (returns long).
static double sqrt(double a)
    Returns the square root of a value.
    The square root of a negative number is not defined in the realm of real numbers, which is why Math.sqrt() returns NaN (Not a Number) when the argument is negative.
    Time Complexity: O(1)
static double cbrt(double a)
    Returns the cube root of a value.
static double hypot(double x, double y)
    Returns the square root of the sum of the squares of `x` and `y`.
static double pow(double a, double b)
    Returns the value of `a` raised to the power of `b`.
static double toRadians(double angdeg)
    Converts an angle in degrees to radians.
static double toDegrees(double angrad)
    Converts an angle in radians to degrees.
static double sin(double a)
    Returns the sine of an angle (in radians).
static double cos(double a)
    Returns the cosine of an angle (in radians).
static double tan(double a)
    Returns the tangent of an angle (in radians).
static double log(double a)
    Returns the natural logarithm (base e) of a value.
static double log10(double a)
    Returns the base-10 logarithm of a value.
static double exp(double a)
    Returns Euler's number `e` raised to the power of a value.
static double signum(double a)
    Returns the sign of a value (1.0 for positive, -1.0 for negative, 0.0 for zero).
static double atan2(double y, double x)
    Returns the angle (in radians) from the origin to the point (x, y).
static double sinh(double x)
    Returns the hyperbolic sine of a value.
static double cosh(double x)
    Returns the hyperbolic cosine of a value.
static double tanh(double x)

Returns the hyperbolic tangent of a value.

Comparable

package java.lang;
public interface **Comparable**<T>
public int compareTo(T o);
The value with the smaller result from compareTo will appear earlier in a sorted order.
`compareTo(b) = 1`
a is considered greater than b, so a will appear after b in a sorted collection.
`compareTo(b) = -1`
a is considered less than b, so a will appear before b in a sorted collection.

Cloneable

package java.lang;
public interface Cloneable 对象需要克隆时实现，重写 Object 类中就有的 clone()方法即可，不声明 Cloneable 调用 clone()方法会抛出 CloneNotSupportedException 异常

Random

package java.lang;
public class **Random** 随机工具
int nextInt(int bound) 返回一个随机整数 [0, n)

Iterable

package java.lang;
public interface **Iterable**<T> 遍历器
default void forEach(Consumer<? super T> action) 每一个元素都执行一次 action

Integer

package java.lang;
public final class **Integer** extends Number implements Comparable<Integer>

public static String toHexString(int i) 返回为无符号整数基数为 16 的整数参数的字符串表示形式 // 170 ==> aa
public int hashCode() hashCode 就是数字本身
public static int parseInt(String s, int radix) throws NumberFormatException 其他进制数转换成 10 进制 // Integer.parseInt("a7a", 16) 16 进制字符串 10 进制数
public static Integer valueOf(String s) throws NumberFormatException 其他类型转数字，null 会宝空指针

```
public static Integer valueOf(int i) {
    if (i >= IntegerCache.low && i <= IntegerCache.high)
        return IntegerCache.cache[i + (-IntegerCache.low)];
    return new Integer(i);
}
```

```
private static class IntegerCache
        static final int low = -128;
        static final int high;
        static final Integer[] cache;       Cache Interger objects.
```

AbstractStringBuilder

```
package java.lang;
abstract class AbstractStringBuilder implements Appendable, CharSequence

static final boolean COMPACT_STRINGS;
@Native static final byte LATIN1 = 0;
@Native static final byte UTF16  = 1;

AbstractStringBuilder(int capacity) {
    if (COMPACT_STRINGS) {
        value = new byte[capacity];
        coder = LATIN1;
    } else {
        value = StringUTF16.newBytesFor(capacity);
        coder = UTF16;
    }
}
public void ensureCapacity(int minimumCapacity) {
    if (minimumCapacity > 0) {
        ensureCapacityInternal(minimumCapacity);
    }
}

private void ensureCapacityInternal(int minimumCapacity) {
    // overflow-conscious code
    int oldCapacity = value.length >> coder;
    if (minimumCapacity - oldCapacity > 0) {
        value = Arrays.copyOf(value,
                newCapacity(minimumCapacity) << coder);
    }
}

private int newCapacity(int minCapacity) {
    int oldLength = value.length;
    int newLength = minCapacity << coder;
    int growth = newLength - oldLength;
    int length = ArraysSupport.newLength(oldLength, growth, oldLength + (2 << coder));
    if (length == Integer.MAX_VALUE) {
        throw new OutOfMemoryError("Required length exceeds implementation limit");
    }
    return length >> coder;
}
```

Number

```
package java.lang;
public abstract class Number implements java.io.Serializable
public abstract long longValue();    获取 long 值
```

Integer

package java.lang;

public final class **Integer** extends Number

implements Comparable<Integer>, Constable, ConstantDesc

```
Integer a = new Interger(12);    // Trigger autoboxing.
Integer b = 12;                  // Won't trigger autoboxing.
int c = 12;
a == b            // true
a == c            // true
```

private static class IntegerCache

Cache to support the object identity semantics of autoboxing for values between -128 and 127 (inclusive) as required by JLS.

See: Interger, Short, Byte, Character, Long

static final int low = -128;

static final int high;

static final Integer[] cache;

static Integer[] archivedCache;

public static int highestOneBit(int i)

Returns an integer with only the highest-order (most significant) bit set to 1, and all other bits set to 0.

This function can be used to find the largest power of two less than or equal to k.

Integer.numberOfLeadingZeros(i) can be used to get the power

```
// Find the largest power of two less than or equal to k
long lower = Long.highestOneBit(k);

// Find the next higher power of two (if any)
long higher = lower == k ? k : lower << 1;
i = 1, highestOneBit(i) = 1
i = 2, highestOneBit(i) = 2
i = 3, highestOneBit(i) = 2
i = 5, highestOneBit(i) = 4
i = 8, highestOneBit(i) = 8
i = 16, highestOneBit(i) = 16
i = 17, highestOneBit(i) = 16
i = 35, highestOneBit(i) = 32
i = 255, highestOneBit(i) = 128
```

public static int lowestOneBit(int i)

public static int numberOfLeadingZeros(int i)

public static int numberOfTrailingZeros(int i)

public static int bitCount(int i)

Returns the number of one-bits in the two's complement binary representation of the specified int value.

This runs in O(log 1)

This function is sometimes referred to as the population count.

public int intValue()

Returns the value of this Integer as an int.

equals

```
public boolean equals(Object obj) {
   if (obj instanceof Integer) {
      return value == ((Integer)obj).intValue();
   }
   return false;
}
```

Long

@jdk.internal.ValueBased
public final class **Long** extends Number
    implements Comparable<Long>, Constable, ConstantDesc

@IntrinsicCandidate
public static int numberOfLeadingZeros(long i)
    Count the number of leading zeros in the binary representation of a given long integer i.
    In other words, it determines the number of consecutive zeros from the leftmost bit (the most significant bit) until the first non-zero bit.

@IntrinsicCandidate
public static int numberOfTrailingZeros(long i)
    Trailing zeros refer to consecutive zeros at the right end of a binary number.

Enum

package java.lang;
public abstract class **Enum**<E extends Enum<E>>
  implements Comparable<E>, Serializable
public static <T extends Enum<T>> T valueOf(Class<T> enumClass, String name)   jpa enum converter

Character

package java.lang;
public final class **Character** implements java.io.Serializable, Comparable<Character>, Constable

public static int compare(char x, char y)
public static char forDigit(int digit, int radix)   Get a char using the given radix. // Character.forDigit((bt[i] & 240) >> 4, 16)
public static Character valueOf(char c)
public static int getNumericValue(char ch)
    Returns the int value that the specified Unicode character represents.
    Example:
```
Character.getNumericValue( '2')      // 2
```

CharSequence

package java.lang;
public interface **CharSequence**
public default IntStream chars()   Convert String to IntStream（ASCII code，0=>48）

Short

String

package java.lang;
public final class **String**
  implements java.io.Serializable, Comparable<String>, CharSequence
        Count the number of substrings:
```
int currentLeftCount = checkCountLine.length()-checkCountLine.replaceAll("\\{","").length();
```

String pool test:

```
String s = "zzc";
String s2 = "zzc";
System.out.println(s == s2);  // true

String s = "zzc";
String s2 = new String("zzc");
System.out.println(s == s2);  // false

String s = "zzc";
String s2 = new String("zzc");
String s3 = s2.intern();
System.out.println(s == s3);  // true
```

public static final Comparator<String> CASE_INSENSITIVE_ORDER = new CaseInsensitiveComparator();

A Comparator that orders String objects as by compareToIgnoreCase. This comparator is serializable.

Note that this Comparator does not take locale into account, and will result in an unsatisfactory ordering for certain locales.

The java.text.Collator class provides locale-sensitive comparison.

@Stable

private final byte[] value;

public String()

The default value is "".

public String(char value[])

Convert character array to string

```
char[] arr=new char[3];
arr[0]='0';
arr[1]='1';
System.out.println("#"+new String(arr)+"#");    // “#01 #”
```

public String(byte bytes[])　　平台默认字符编码，将字节数组解码为字符串

public String(byte bytes[], int offset, int length)　　平台默认字符编码，将字节数组解码为字符串，从索引 0 开始，长度 10

public String(byte[] bytes, int offset, int length)

Constructs a new String by decoding the specified subarray of bytes using the platform's default charset.

The length of the new String is a function of the charset, and hence may not be equal to the length of the subarray.

Params:

bytes – The bytes to be decoded into characters

offset – The index of the first byte to decode

length – The number of bytes to decode

Throws:

IndexOutOfBoundsException – If offset is negative, length is negative, or offset is greater than bytes.length - length

public String(byte bytes[], String charsetName) throws UnsupportedEncodingException　　指定编码的字符串

public static String valueOf(Object obj)　　其他类型转换成字符串

public static String format(String format, Object... args)　返回格式字符串（%s %.2f %d 填充占位符）

```
// %-3.2f      保留小数点 3 为，不算小数点总共 3 位，超出就超出显示，不会截断
// %-3.3s      字符串为截断长度，即使传入的是数字，字符串也是固定长度
// 6.2f%%    转义显示百分号：%%
```

public static String join(CharSequence delimiter, CharSequence... elements)

public static String join(CharSequence delimiter, Iterable<? extends CharSequence> elements)

Join a character list

```
        StringBuilder sb = new StringBuilder();
        for (Character c : lls) {
            sb.append(c);
        }
        String joinedString = sb.toString();
```

Join a string list

```
        String result = String.join(", ", list);
```

Join a char[] array in Java with a delimiter

```
        char[] chars = {'a', 'b', 'c', 'd', 'e'};
        StringBuilder result = new StringBuilder();

        for (int i = 0; i < chars.length; i++) {
            result.append(chars[i]);
            if (i < chars.length - 1) {
                result.append(", "); // Add delimiter
            }
        }
```

public int length()
public boolean equals(Object anObject)
public boolean isEmpty()
public boolean matches(String regex)
public boolean contains(CharSequence s)
public int indexOf(String str)
public int indexOf(String str, int fromIndex)
public int lastIndexOf(String str)
public int hashCode()
根据字符和字符串长度计算得出， s[0]*31^(n-1) + s[1]*31^(n-2) +...+ s[n-1]

public char charAt(int index)
Time Complexity: O(1)
public String substring(int beginIndex, int endIndex) 获取一个子字符串，endIndex 不能超出范围 [0, 2)
In Java 7 Update 6 and Later, substring() **creates a new character array** that holds the copied portion of the original string.
The **time complexity** is O(m), where m is the length of the substring (endIndex - startIndex).
String s = new String("dddd");
System.out.println("#"+s.substring(0,0)+"#"); // ##
public String substring(int beginIndex) 获取一个子字符串，包含到最后一个字符
public char[] toCharArray() 转换成字符数组并返回
Time Complexity: O(n)
public byte[] getBytes() 平台默认字符编码 字符串编码为字节数组
public byte[] getBytes(String charsetName) 使用指定字符编码，字符串编码成字节数组

public String toUpperCase() 转换成大写
public String toLowerCase() 转换成小写
public String toUpperCase(Locale locale) 转换成大写，指定语言 // "test".toUpperCase(Local.ENGLISH)
Time Complexity: O(n)
public boolean startsWith(String prefix) 开头的字符串是否为 prefix，大小写敏感
public boolean endsWith(String suffix) 结尾的字符串是否为 suffix
public boolean equalsIgnoreCase(String anotherString) 比较字符串是否相同，不区分大小写
public int compareTo(String anotherString) 比较字符串大小（等于返回 0 覆盖，小于返回-n，大于返回 n）
public String trim() 得到去掉左右空格的字符串
public String[] split(String regex) 根据符号拆分字符串

public String[] split(String regex, int limit) 根据符号拆分字符串

当 limit>0 切割次数=limit-1,因此当 limit=1 时，不切割，原字符串输出

当 limit=0 与不写 limit 一致：不限制切割次数，出现几次匹配切割几次。但是去除最后空字符串

当 limit<0 不限制切割次数，出现几次匹配切割几次，不去除最后空字符串。

public String concat(String str) 拼接字符串，不修改原字符串

the concat string in the parameters must not be null

public String repeat(int count)

public String replace(char oldChar, char newChar)

Time Complexity: O(n)

public String replace(CharSequence target, CharSequence replacement)

This method replaces all occurrences of the specified target sequence with the replacement sequence.

Treated as a literal string, not a regular expression. It doesn't recognize special regex characters like . or *.

public String replaceAll(String regex, String replacement)

Time Complexity: O(n)

This method replaces all substrings of the string that match the given regular expression with the specified replacement.

replacement 内可以通过 $1 获取前方正则分组匹配的内容，$0 为前方全部匹配内容

replacement 是普通字符串，转义字符用一个斜杠就好 \t

```
// public static void main(String[] args) throws UnknownHostException
// (.*\()|(\))  ==>  String[] args throws UnknownHostException
// configPath.replaceAll("\\\\","\\\\\\\\");  C:\\Users\\saidake\\.smp
```

Matcher.quoteReplacement(String s)

Escapes characters in a replacement string that have special meanings in replacement patterns.

When using String.replaceAll() or Matcher.replaceAll(),

It ensures that characters such as $ (which is used for group references) and \ (escape character) are treated as literal values in replacement operations.

Pattern.quote(String s)

Escapes all special regex characters in a given string so that it can be used as a literal pattern.

It ensures that all regex metacharacters (. ^ $ * + ? { } [ ] \ | ( )) are treated as literal characters in the pattern.

Escape String

When printing a regex string, it displays as "*".

However, the actual regex in memory should be "*", since the backslash "\" is an escape character in Java strings, but the asterisk "*" is not.

Original File: "*"

Print result of read String: "*" (iteral)

Pass the read string as a regex parameter: "*" (iteral and valid)

public native String intern()

当调用 intern 方法时，如果池已经包含一个等于此 String 对象的字符串（用 equals 方法确定），则返回池中的字符串。

否则，将此 String 对象添加到池中，并返回此 String 对象的引用。

Boolean

package java.lang;

public final class **Boolean** implements java.io.Serializable, Comparable<Boolean>

public static final Boolean TRUE = new Boolean(true);

public static final Boolean FALSE = new Boolean(false);

public static boolean getBoolean(String name)　当且仅当以参数命名的系统属性存在，且等于 “true” 字符串时，才返回 true。

StringBuilder

package java.lang;
public final class **StringBuilder**　　线程不安全，效率高，数组初始容量是 16，数组动态扩容
　extends AbstractStringBuilder
　implements java.io.Serializable, CharSequence

public StringBuilder() {
　super(16);
}

StringBuffer

package java.lang;
public final class **StringBuffer**
　　A thread-safe, mutable sequence of characters. A string buffer is like a String, but can be modified.
　　At any point in time it contains some particular sequence of characters, but the length and content of the sequence can be changed through certain method calls.

@IntrinsicCandidate
public StringBuffer() {
　super(16);
}
public StringBuffer(int capacity)　　指定字符容量，超出自动增加容量
public StringBuffer(String str)　　当前字符串 加 16 字符　超出自动增加容量

public synchronized int length()
public synchronized int capacity()

public synchronized StringBuffer append(Object obj)　　将各种数据转换成字符串追加到当前对象中
public synchronized char charAt(int index)　　得到索引 n 上的字符
public synchronized void setCharAt(int index, char ch)　　index 处字符用 ch 替换
public synchronized StringBuffer delete(int start, int end)　　从当前对象中删除字符串
public synchronized StringBuffer deleteCharAt(int index)　　删除索引处的字符

public synchronized StringBuffer insert(int offset, Object obj)　字符串插入索引 index 处
public synchronized StringBuffer reverse()　　翻转字符串
public synchronized StringBuffer replace(int start, int end, String str)　　用 str 替换指定内容[start, end)
public synchronized String toString()　　转换成字符串　//没有 contains 方法

public int indexOf(String str)　　返回子串索引（传入字符会保错）
public int lastIndexOf(String str)　　返回子串逆序索引（传入字符会报错）

public synchronized void ensureCapacity(int minimumCapacity)　　确保最低容量
public synchronized void getChars(int srcBegin, int srcEnd,　char[] dst, int dstBegin)　　返回 srcBegin 到 srcEnd 的字符到 dst
public synchronized CharSequence subSequence(int start, int end)　　返回字符子序列;

public synchronized String substring(int start)　　返回子串；
public synchronized String substring(int start, int end)　返回子串；
public synchronized void trimToSize()　　缩小 value 的容量到真实内容大小；

(Thread)

package java.lang;
public interface **Runnable**　函数式接口，可执行内容
public abstract void run();　执行线程内容

Thread

package java.lang;
public class **Thread** implements Runnable

public enum State {
  NEW,
    A thread that has been created but has not yet started.
    The thread is in this state when it is instantiated using the new keyword but before the start() method is called.
    No resources have been allocated for the thread's execution.
  RUNNABLE,
    A thread that is ready to run and may be running or waiting for CPU time.
    When the start() method is called, the thread transitions to the RUNNABLE state.
    The thread can be in this state either actively running or waiting for CPU resources.

  BLOCKED,
    A thread that is blocked waiting for a monitor lock to enter or re-enter a synchronized block/method.
    This state occurs when a thread tries to enter a synchronized block or method but another thread holds the lock.
    The thread will remain in this state until the lock is released.
  WAITING,
    A thread that is waiting indefinitely for another thread to perform a particular action.
    This state occurs when methods like Object.wait(), Thread.join(), or LockSupport.park() are called without a timeout.
    The thread remains in this state until it is notified or interrupted.
  TIMED_WAITING,
    A thread that is waiting for another thread to perform a specific action for a specified period.
    This state occurs when methods like Thread.sleep(long millis), Object.wait(long timeout), or Thread.join(long millis) are called with a timeout.
    The thread will transition out of this state after the timeout period elapses or if it is notified/interrupted.

  TERMINATED;　　has completed its execution
}

public Thread(Runnable target)
public static native Thread currentThread();
public static native boolean holdsLock(Object obj);
    This static method checks if the current thread holds the monitor lock on the specified object.
public void run()　　This would run the code in the current thread, not in a new one
private void exit()
public synchronized void start()　　Starts a new thread, which runs the code in parallel

private void init(ThreadGroup g, Runnable target, String name, long stackSize, AccessControlContext acc, boolean

inheritThreadLocals)

public final void setPriority(int newPriority)

This method changes the priority of the thread.

The priority is an integer value between Thread.MIN_PRIORITY (1) and Thread.MAX_PRIORITY (10), with Thread.NORM_PRIORITY (5) as the default.

public final int getPriority()

final native void wait(long timeout)

This method causes the current thread to wait until another thread invokes notify() or notifyAll() on the same object. (WAITING)

The thread releases the lock on the object and enters the waiting state.

static native void sleep(long millis)

This static method pauses the execution of the current thread for the specified number of milliseconds. (TIMED-WATING)

final void join() throws InterruptedException

This method waits for the thread on which it's called to die before proceeding with the next execution step in the current thread.

```
thread.join(); // Waits for the thread to finish
```

static native void yield();

This static method causes the current thread to temporarily pause its execution and allow other threads of the same priority to execute.

It's a hint to the thread scheduler that the current thread is willing to yield its current use of the CPU.

If there are no other threads competing for the CPU, the current thread will remain largely unaffected by the call to Thread.yield() and will likely continue to execute.

```
Thread.yield(); // Yield execution to other threads
```

static int activeCount()

static int enumerate(Thread tarray[])

static native Thread currentThread()

public ClassLoader getContextClassLoader()

void interrupt()

This method interrupts the thread on which it's called. If the thread is in a blocked state (like waiting or sleeping), it will throw an InterruptedException.

```
thread.interrupt(); // Interrupt the thread
```

Interrupting a Thread:

When you call interrupt() on a thread, it sets an internal flag called the interrupt status. This flag serves as a signal to the thread that it has been interrupted.

However, calling interrupt() does not immediately change the state of the thread.

The thread continues its execution unless it's in a state where it can respond to the interruption, like when it's waiting, sleeping, or blocked.

Effect on Running Threads:

Running threads are not directly affected by interrupt().

The thread does not stop, terminate, or change state just because it's interrupted. It only changes the internal interrupt flag to indicate that an interruption request was made.

The thread itself must check this flag and handle the interruption appropriately.

Effect on Sleeping Threads (sleep()):

If a thread is in the TIMED_WAITING state due to a call to sleep(), and interrupt() is called, the thread will

immediately throw an InterruptedException.

This exception interrupts the sleep, and the thread exits the TIMED_WAITING state early. The interrupt flag is also cleared when the exception is thrown.

Methods Throwing InterruptedException:

Many methods that throw InterruptedException (like Thread.sleep(long millis), wait(), join(), etc.) automatically clear the interrupt flag when they throw this exception.

This means that if you check the interrupt status using isInterrupted() after catching InterruptedException, it will return false because the flag was cleared.

Using isInterrupted() to Check Interrupt Status:

The interrupt status is an inherent flag within a thread. You can use the isInterrupted() method to check if a thread has been interrupted.

This can be used to gracefully stop a thread's execution.

For example, if you want to stop a thread, you might call thread.interrupt(), and within the thread's run() method, you can periodically check thread.isInterrupted() to decide whether to terminate the thread gracefully.

```
public class InterruptExample implements Runnable {
    @Override
    public void run() {
        try {
            while (!Thread.currentThread().isInterrupted()) {
                System.out.println("Thread is running...");
                Thread.sleep(1000); // Simulate work
            }
        } catch (InterruptedException e) {
            // Handle the interruption
            System.out.println("Thread was interrupted!");
            // Restore the interrupted status if needed
            Thread.currentThread().interrupt();
        }
    }

    public static void main(String[] args) throws InterruptedException {
        Thread thread = new Thread(new InterruptExample());
        thread.start();

        Thread.sleep(3000); // Main thread sleeps for 3 seconds
        thread.interrupt(); // Interrupt the thread
    }
}
```

final native boolean isAlive();

final boolean isDaemon()

final void setDaemon(boolean on)

final synchronized void setName(String name)

Thread Synchronization

Synchronized Blocks and Methods

- Synchronized Instance Methods
- Synchronized Static Methods
- Synchronized Blocks Within Methods

Locks (java.util.concurrent.locks.Lock)

Java's java.util.concurrent.locks package provides more advanced synchronization mechanisms like Lock and ReentrantLock, which offer more flexibility than synchronized.

Condition Variables (java.util.concurrent.locks.Condition)

Condition variables are used for inter-thread signaling, where one thread waits for a condition to be true and another thread signals when that condition is met.

Thread Management

CountDownLatch, CyclicBarrier, Semaphore, Phaser

Volatile Keyword

The volatile keyword ensures visibility of changes to variables across threads. This means that changes made by one thread to a volatile variable are visible to other threads.

```
private volatile boolean flag = false;

public void setFlagTrue() {
    flag = true;
}

public void checkFlag() {
    while (!flag) {
        // Wait until flag is true
    }
    // Proceed when flag is true
}
```

Atomic Variables (java.util.concurrent.atomic)

Atomic variables like AtomicInteger, AtomicBoolean, etc., provide a lock-free mechanism for thread-safe operations on single variables.

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

public void increment() {
    counter.incrementAndGet();
}
```

Thread Execution

Extends Thread class

```
public class MyThread extends Thread {
  @Override public void run() {
    System.out.println("I am a thread!");
  }
}

MyThread thread = new MyThread();
thread.start();
```

Implements Runnable interface

```
public class MyRunnable implements Runnable {
  @Override public void run() {
    System.out.println("I am a thread!");
  }
}

MyRunnable runnable = new MyRunnable();
Thread thread = new Thread(runnable);
thread.start();
```

Implements Callable interface

```
import java.util.concurrent.Callable;
import java.util.concurrent.FutureTask;
public class CallableExample implements Callable<String> {
  @Override
  public String call() throws Exception {
    // This code will be executed in a separate thread.
    return "Hello from a new thread!";
  }
  public static void main(String[] args) throws Exception {
    // Create a Callable object.
    Callable<String> callable = new CallableExample();

    // Create a FutureTask object.
    FutureTask<String> futureTask = new FutureTask<>(callable);
```

```
        // Create a Thread object and pass the FutureTask object to the constructor.
        Thread thread = new Thread(futureTask);
        // Start the thread.
        thread.start();
        // Get the result from the FutureTask object.
        String result = futureTask.get();
        // Print the result.
        System.out.println(result);
    }
}
```

The Executors class provides a number of methods for creating and managing threads.

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;


public class Main {
  public static void main(String[] args) {
    ExecutorService executorService = Executors.newFixedThreadPool(10);

    Runnable task = () -> {
      System.out.println("Hello from the thread pool!");
    };

    executorService.submit(task);
    executorService.shutdown();
  }
}
```

The ThreadPoolExecutor class provides a way to create a pool of threads that can be used to execute tasks.

```
import java.util.concurrent.Executors;
import java.util.concurrent.ThreadPoolExecutor;

public class Main {
  public static void main(String[] args) {
    // Create a thread pool with 10 core threads and a maximum of 20 threads
    ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(10);

    // Create a new task to be executed by the thread pool
    Runnable task = () -> {
      System.out.println("Hello from the thread pool!");
    };

    // Submit the task to the thread pool
    executor.submit(task);

    // Shut down the thread pool
    executor.shutdown();
  }
}
```

The ScheduledExecutorService class provides a way to schedule tasks to be executed at a specific time or interval.

```
ScheduledExecutorService scheduledExecutorService = Executors.newScheduledThreadPool(1);
scheduledExecutorService.schedule(new Runnable() {
  public void run() {
  System.out.println("Hello, world!");
  }
}, 5, TimeUnit.SECONDS);
```

Process scheduling algorithm (google)

Process scheduling algorithms are used by the CPU to select one process from many processes for execution, maximizing CPU utilization by increasing throughput.

1) Round Robin (RR):
   - Description: Each process is assigned a fixed time slice (quantum) in a cyclic order.
   - Characteristics: Preemptive; fair and good for time-sharing systems but performance depends on the quantum size.
2) Priority Scheduling:
   - Description: Each process is assigned a priority, and the process with the highest priority is scheduled next.
   - Characteristics: Can be preemptive or non-preemptive; risk of starvation for low-priority processes (mitigated by aging).
3) Lottery Scheduling:
   - Description: Processes are given "lottery tickets" for CPU time, and a random ticket is drawn to select the next process.
   - Characteristics: Probabilistic and fair; can be fine-tuned by adjusting the number of tickets for each process.
4) First-Come, First-Served (FCFS):
   - Description: Processes are scheduled in the order they arrive in the ready queue.
   - Characteristics: Simple, non-preemptive; can lead to long waiting times (convoy effect).

5) Shortest Job Next (SJN) or Shortest Job First (SJF):
   - Description: The process with the shortest execution time is selected next.
   - Characteristics: Can be preemptive or non-preemptive; optimal in terms of minimizing average waiting time but requires knowledge of future process lengths.
6) Shortest Remaining Time First (SRTF):
   - Description: Preemptive version of SJF, where the process with the shortest remaining execution time is selected next.
   - Characteristics: Optimal for minimizing average waiting time but requires accurate estimation of process burst times.

7) Multilevel Queue Scheduling:
   - Description: Processes are divided into multiple queues based on priority or type, with each queue having its own scheduling algorithm.
   - Characteristics: Suitable for systems with distinct process groups; allows different scheduling policies for different types of processes.
8) Multilevel Feedback Queue Scheduling:
   - Description: Similar to multilevel queue scheduling but allows processes to move between queues based on their behavior and age.
   - Characteristics: Dynamic adjustment of process priority; reduces the risk of starvation and adapts to varying workloads.

9) Highest Response Ratio Next (HRRN):
   - Description: Prioritizes processes based on the ratio of waiting time plus service time to service time.
   - Characteristics: Balances short and long processes; reduces the risk of starvation and improves average waiting time.
10) Fair-Share Scheduling:
   - Description: Allocates CPU time based on the allocation of shares to users or groups rather than individual processes.
   - Characteristics: Ensures fair distribution of CPU time among users or groups; can be complex to implement.

ThreadLocal

package java.lang;
public class **ThreadLocal**<T>

The ThreadLocal class in Java provides a way to store data that is local to a specific thread.
ThreadLocal variable is shared among threads, meaning the variable itself is the same for all threads.
In other words, all threads can access the same ThreadLocal variable.
However, each thread has its own independent value for that variable.

Managing ThreadLocal variables in a multi-threaded environment, especially with thread pools, requires careful consideration.
Manually setting and clearing ThreadLocal values or using task decorators can help prevent data leakage and ensure that each task runs with the correct context.

Shared Heap but Isolated Values:

Even though all threads share the heap memory, each thread's ThreadLocalMap ensures that the data is isolated for each thread.
The values stored in the ThreadLocalMap are on the heap, but the key to access them (ThreadLocal object) is specific to the thread.
This prevents one thread from accessing another thread's ThreadLocal data.
Because ThreadLocal variables are isolated to individual threads, they avoid the need for explicit synchronization when used within a single thread.

Thread Reuse:

In a thread pool, threads are reused to execute different tasks.
This reuse can cause issues with ThreadLocal because the data from a previous task could persist and be accessed by a new task running on the same thread.
If not handled properly, ThreadLocal variables can lead to unintended data sharing between different tasks executed by the same thread.

Automatic Cleanup:

When a thread terminates, its ThreadLocal object are generally cleaned up if the ThreadLocal objects are no longer referenced.
However, values might persist if there are other strong references.

Manual Removal

To prevent memory leaks and ensure proper cleanup, you should manually call ThreadLocal.remove() to clear the value associated with a ThreadLocal,
especially in environments with thread pools where threads are reused.

Usage Scenarios

User Sessions:

Each request in a web application needs to access user session information specific to that thread.
Usage:

Store user session data in a ThreadLocal to ensure that session information is kept separate across different threads.

Session Context:

Spring Security maintains user authentication and session information using the SecurityContextHolder, which by default is backed by a ThreadLocal.
This allows thread-local storage of security context data such as the authenticated user.

Logging Context:

Maintain contextual information for logging purposes, such as request IDs or user IDs, across various parts of the application.
Usage:

Store logging context data in ThreadLocal to include contextual information in log entries.

Transaction Management:

Manage transaction states for each thread, ensuring that transactions are handled independently.

Usage:

Store the transaction state in ThreadLocal to ensure that transaction management is thread-local.

Spring Transaction Management:

Spring uses ThreadLocal to manage transactional context. When a transaction is started, Spring stores the transaction status and context in a ThreadLocal variable.

This allows Spring to track the transaction across various method calls within the same thread.

Configuration Settings:

Use different configuration settings or parameters specific to different threads.

Usage:

Store configuration settings in ThreadLocal so that each thread can have its own configuration settings.

static class ThreadLocalMap

In the ThreadLocal class, the ThreadLocalMap is an internal data structure used to store thread-local values for each thread.

It maintains a map of ThreadLocal instances to their corresponding values.

Each thread has its own ThreadLocalMap instance, and the map is not shared between threads.

Key and Value

The key for the ThreadLocalMap in the ThreadLocal class is an instance of ThreadLocal with WeakReference.

Each ThreadLocal object has a unique instance, and this uniqueness is leveraged to store and retrieve values in a per-thread map.

Notes

The threads in the thread pool will not be released, so even if the ThreadLocal object is reclaimed by the gc, its ThreadLocalMap instance will not be released,

so we need to call the remove method to manually clean up the object.

```
static class Entry extends WeakReference<ThreadLocal<?>> {
    Object value;  // The value associated with this ThreadLocal.
    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v;
    }
}
private Entry[] table;
```

public void set(T value)

Sets the value for the current thread's ThreadLocal variable.

public T get()

Retrieves the current thread's value for the ThreadLocal variable. If the thread does not have a value, it will invoke the initialValue() method to create one.

public void remove()

Removes the current thread's value for the ThreadLocal variable. This is useful for cleaning up after a thread is done with its work.

This operation clears the strong reference to the value held in the ThreadLocalMap, making the value eligible for garbage collection if no other references to it exist.

ThreadLocalMap getMap(Thread t)

protected T initialValue()

Usage

```
public class ThreadLocalExample {

    // Create a ThreadLocal variable
    private static final ThreadLocal<Integer> threadLocalValue = ThreadLocal.withInitial(() -> 1);

    public static void main(String[] args) {
        // Start two threads
        Thread thread1 = new Thread(() -> {
            Integer value = threadLocalValue.get();
            System.out.println("Thread 1 initial value: " + value);
            threadLocalValue.set(value + 1);
            System.out.println("Thread 1 updated value: " + threadLocalValue.get());
        });

        Thread thread2 = new Thread(() -> {
            Integer value = threadLocalValue.get();
            System.out.println("Thread 2 initial value: " + value);
            threadLocalValue.set(value + 10);
            System.out.println("Thread 2 updated value: " + threadLocalValue.get());
        });

        thread1.start();
        thread2.start();
    }
}
```

Using Decorators

Wrap your tasks in a decorator that sets the ThreadLocal value before the task runs and clears it afterward.

```
public class ThreadLocalTaskDecorator implements Runnable {
    private final Runnable task;
    private final String value;

    public ThreadLocalTaskDecorator(Runnable task, String value) {
        this.task = task;
        this.value = value;
    }

    @Override
    public void run() {
        try {
            threadLocal.set(value);
            task.run();
        } finally {
            threadLocal.remove();
        }
    }
}

// Usage:
ExecutorService executor = Executors.newSingleThreadExecutor();
executor.submit(new ThreadLocalTaskDecorator(() -> {
    // Task logic here
}, "Some value"));
```

ContextCopyingDecorator

If you're using Spring, the TaskDecorator interface allows you to customize the execution of tasks, which you can use to handle ThreadLocal.

```
public class ContextCopyingDecorator implements TaskDecorator {
    @Override
    public Runnable decorate(Runnable runnable) {
        String context = threadLocal.get();
```

```
        return () -> {
            try {
                threadLocal.set(context);
                runnable.run();
            } finally {
                threadLocal.remove();
            }
        };
    }
}

@Bean
public ThreadPoolTaskExecutor taskExecutor() {
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
    executor.setTaskDecorator(new ContextCopyingDecorator());
    executor.initialize();
    return executor;
}
```

InheritableThreadLocal

package java.lang;

public class **InheritableThreadLocal**<T> extends ThreadLocal<T>

Transfer or propagate ThreadLocal values from the parent thread to the child thread

Values are inherited only when the thread is created.

Once the thread is created, changes to the parent's InheritableThreadLocal value will not reflect in the child thread.

TransmittableThreadLocal: Values are manually propagated to threads, even in reused threads, through wrappers like TtlRunnable.

protected T childValue(T parentValue)

ThreadLocalMap getMap(Thread t)

Returns the inheritableThreadLocals of the current thread, which will get the value of the parent thread when thread.init is initialized

void createMap(Thread t, T firstValue)

Create a new ThreadLocalMap like ThreadLocal class

(Exception)

ArithmeticException

package java.lang;

public class **ArithmeticException** extends RuntimeException 1/0

Throwable

package java.lang;

public class **Throwable** implements Serializable Throwable 是 Java 语言中所有错误或异常的超类。下一层分为 Error 和 Exception

public String getLocalizedMessage() 以用户的本地语言(中文，日语等)返回异常的名称。

public String getMessage() 获取详细信息

public synchronized Throwable getCause() 获取原因

例如线程池抛出的异常，最外层是 ExecutionException，所以 getCause() 能够获取到里层 RuntimeException；

直接捕获抛出的异常，调用 getCause() 则返回的 null；

public final synchronized Throwable[] getSuppressed() 获取压制的异常

public void printStackTrace(PrintWriter s)　打印过程从外向内，最外层抛出的位置最先打印

Exception

package java.lang;

public class **Exception** extends Throwable

Determine whether the exception is an unchecked exception or a checked exception by checking whether the exception will interrupt the entire business logic.

1. Unchecked Exceptions (Runtime Exceptions)

   Unchecked exceptions, also known as runtime exceptions, are not checked at compile-time.

   Often indicate programming errors or conditions that are outside the control of the program (e.g., invalid user input).

   - RuntimeException is the parent class of all unchecked exceptions.

   Example:

   - NullPointerException

     Thrown when an application attempts to use null in a case where an object is required.

     ```
     String str = null;
     int length = str.length(); // Throws NullPointerException
     ```

   - ClassCastException

     Thrown to indicate that the code has attempted to cast an object to a subclass of which it is not an instance.

     ```
     Object x = new Integer(0);
     String s = (String) x; // Throws ClassCastException
     ```

   - TypeNotPresentException

     Thrown when an application tries to access a type that is not present in the current classpath.

     ```
     try {
         Class<?> clazz = Class.forName("com.example.NonExistentClass");
     } catch (ClassNotFoundException e) {
         throw new TypeNotPresentException("com.example.NonExistentClass", e);
     }
     ```

   - ArithmeticException

     Thrown when an exceptional arithmetic condition has occurred, such as divide by zero.

     ```
     int result = 1 / 0; // Throws ArithmeticException
     ```

   - NumberFormatException

     Thrown to indicate that an attempt to convert a string to a numeric type has failed.

     ```
     int number = Integer.parseInt("abc"); // Throws NumberFormatException
     ```

   - ArrayIndexOutOfBoundsException

     Thrown to indicate that an array has been accessed with an illegal index.

     ```
     int[] array = new int[5];
     int value = array[10]; // Throws ArrayIndexOutOfBoundsException
     ```

   - ArrayStoreException

     Thrown to indicate that an attempt has been made to store the wrong type of object into an array of objects.

     ```
     Object[] array = new String[5];
     array[0] = 1; // Throws ArrayStoreException
     ```

   - UnsupportedOperationException

     Thrown to indicate that the requested operation is not supported.

     ```
     List<String> immutableList = Collections.unmodifiableList(new ArrayList<>());
     immutableList.add("test"); // Throws UnsupportedOperationException
     ```

   - IndexOutOfBoundsException

     Thrown to indicate that an index of some sort (such as to an array, to a string, or to a vector) is out of range.

     ```
     String str = "example";
     char ch = str.charAt(10); // Throws IndexOutOfBoundsException
     ```

- AssertionError

  Use assertions to check for conditions that should never happen, not for regular error handling.

  Assertions are generally used during development and testing to catch bugs. They are not typically enabled in production environments.

  ```
  int value = 10;
  assert value > 15 : "Value should be greater than 15";
  ```

- SecurityException

  Thrown by the security manager to indicate a security violation.

  ```
  System.setSecurityManager(new SecurityManager());
  System.exit(1); // Throws SecurityException
  ```

- IllegalArgumentException

  Thrown to indicate that a method has been passed an illegal or inappropriate argument.

  ```
  Thread.sleep(-1000); // Throws IllegalArgumentException
  ```

- IllegalStateException

  Thrown to indicate that a method has been invoked at an illegal or inappropriate time.

  ```
  List<String> list = new ArrayList<>();
  Iterator<String> iterator = list.iterator();
  iterator.next(); // Throws IllegalStateException
  ```

2. Checked Exceptions:

   Checked exceptions are the exceptions that are checked at compile-time.

   This means the compiler ensures that the code handles these exceptions either by catching them in a try-catch block or by specifying that the method throws these exceptions using the throws keyword.

   Some key points about checked exceptions:

   - Except for RuntimeException and Error, all exceptions belong to the checked exceptions
   - A Class that inherits Exception class will become a Checked Exception.

   Example:

   - IOException

     Thrown when an I/O operation fails or is interrupted.

     ```
     try {
         FileInputStream file = new FileInputStream("example.txt");
     } catch (IOException e) {
         e.printStackTrace();
     }
     ```

   - FileNotFoundException

     Thrown when an attempt to open the file denoted by a specified pathname has failed.

     ```
     try {
         FileInputStream file = new FileInputStream("nonexistent.txt");
     } catch (FileNotFoundException e) {
         e.printStackTrace();
     }
     ```

   - ClassNotFoundException

     Thrown when an application tries to load a class through its string name but no definition for the class with the specified name could be found.

     ```
     try {
         Class.forName("com.example.NonExistentClass");
     } catch (ClassNotFoundException e) {
         e.printStackTrace();
     }
     ```

   - SQLException

     Thrown when there is an error accessing the database or other database access errors.

     ```
     try {
         Connection connection = DriverManager.getConnection("jdbc:mysql://localhost/db",
     "user", "password");
     } catch (SQLException e) {
         e.printStackTrace();
     ```

```
    }
```

- InterruptedException

  Thrown when a thread is waiting, sleeping, or otherwise occupied, and the thread is interrupted, either before or during the activity.

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    e.printStackTrace();
}
```

- NoSuchMethodException

  Thrown when a particular method cannot be found.

```
try {
    Method method = MyClass.class.getMethod("nonExistentMethod");
} catch (NoSuchMethodException e) {
    e.printStackTrace();
}
```

- InvocationTargetException

  Thrown to indicate that an exception occurred while attempting to invoke a method via reflection.

```
try {
    Method method = MyClass.class.getMethod("myMethod");
    method.invoke(new MyClass());
} catch (InvocationTargetException e) {
    e.printStackTrace();
}
```

- MalformedURLException

  Thrown to indicate that a malformed URL has occurred.

```
try {
    URL url = new URL("malformed:url");
} catch (MalformedURLException e) {
    e.printStackTrace();
}
```

- CloneNotSupportedException

  Thrown to indicate that the clone method in class Object has been called to clone an object, but that the object's class does not implement the Cloneable interface.

```
try {
    MyClass obj = new MyClass();
    MyClass clonedObj = (MyClass) obj.clone();
} catch (CloneNotSupportedException e) {
    e.printStackTrace();
}
```

- InstantiationException

  Thrown when an application tries to create an instance of a class using the newInstance method in class Class, but the specified class object cannot be instantiated because it is an interface or is an abstract class.

```
try {
    MyClass obj = MyClass.class.newInstance();
} catch (InstantiationException e) {
    e.printStackTrace();
}
```

- IllegalAccessException

  Thrown when an application tries to reflectively create an instance (other than an array), set or get a field, or invoke a method, but the currently executing method does not have access to the definition of the specified class, field, method, or constructor.

```
try {
    MyClass obj = MyClass.class.newInstance();
```

```
                    } catch (IllegalAccessException e) {
                        e.printStackTrace();
                    }
```

Error

package java.lang;

public class **Error** extends Throwable　　Error 类是指 java 运行时系统的内部错误和资源耗尽错误。应用程序不会抛出该类对象。如果 出现了这样的错误，除了告知用户，剩下的就是尽力使程序安全的终止。

VirtualMachineError　虚拟机错误

StackOverflowError

OutOfMemoryError

UnknownError

InternalError

IOError　　io 错误

AssertionError　断言错误

ThreadDeath　　线程死亡（强制将运行的线程停止）

AnnotationFormatError　注解解析错误

AWTError　　发生严重的 Abstract（A） Window（W） Toolkit（T） 错误

OutOfMemoryError

package java.lang;

public class **OutOfMemoryError** extends VirtualMachineError

java.lang.OutOfMemoryError: Java heap space

Creating too many objects or very large objects that exceed the available heap space.

Holding large data structures (e.g., huge arrays or collections) can quickly consume available memory.

Caches that grow indefinitely without limits can exhaust memory.

Increase Heap Size:　Use the -Xmx flag to increase the maximum heap size.

Memory Leak Detection:　Use profiling tools like VisualVM, YourKit, or Eclipse MAT to identify memory leaks.

Optimize Memory Usage:　Review and optimize the application code to reduce memory consumption.

java.lang.OutOfMemoryError: Direct buffer memory

The JVM has run out of direct buffer memory, used by NIO (New I/O) for direct byte buffers.

Caches that grow indefinitely without limits can exhaust memory.

Increase Direct Memory Size:　Use the -XX:MaxDirectMemorySize flag to increase the maximum direct memory size.

Optimize Direct Buffer Usage:　Ensure direct buffers are properly released and reused.

java.lang.OutOfMemoryError: Metaspace

For Java 8 and later, excessive class metadata or a large number of dynamically generated classes can exhaust the Metaspace area.

Excessive use of reflection can lead to high memory consumption, particularly if many classes are dynamically loaded.

Increase Metaspace Size:　Use the -XX:MaxMetaspaceSize flag to increase the maximum Metaspace size.

Reduce Loaded Classes:　Optimize the application to load fewer classes, possibly by reducing the number of libraries or using class unloading.

java.lang.OutOfMemoryError: PermGen space (Pre-Java 8)

The Permanent Generation (PermGen) space, where class metadata was stored in versions prior to Java 8, is exhausted.

Increase PermGen Size:　Use the -XX:MaxPermSize flag to increase the PermGen space.

Upgrade to Java 8 or Later:　Java 8 and later use Metaspace instead of PermGen, which often mitigates this issue.

java.lang.OutOfMemoryError: unable to create new native thread

The JVM cannot create a new thread due to resource constraints, often because of a limit on the number of threads or insufficient memory.

Threads that are created but not properly managed or terminated can consume memory resources.

Increase Thread Limits: Increase the limit on the number of threads at the OS level.

Reduce Thread Usage: Optimize the application to use fewer threads, such as using thread pools.

Increase JVM Memory: Ensure there is sufficient memory available for thread stack space.

java.lang.OutOfMemoryError: GC overhead limit exceeded

The JVM is spending too much time performing garbage collection with little memory being freed.

Increase Heap Size: Use the -Xmx flag to increase the maximum heap size.

Optimize Memory Usage: Identify and fix memory leaks or reduce memory usage.

Tweak GC Settings: Adjust GC settings to better handle the application's memory allocation patterns.

java.lang.OutOfMemoryError: Compressed class space

The JVM has exhausted the space allocated for compressed class pointers.

Increase Compressed Class Space: Use the -XX:CompressedClassSpaceSize flag to increase the size of the compressed class space.

Optimize Class Loading: Reduce the number of loaded classes and ensure proper class unloading.

java.lang.OutOfMemoryError: Out of swap space?

The system has run out of swap space, which might be due to excessive memory usage or improper swap configuration.

Increase Swap Space: Increase the amount of swap space on the system.

Reduce Memory Usage: Optimize the application to use less memory.

java.lang.OutOfMemoryError: Requested array size exceeds VM limit

An attempt was made to allocate an array larger than the JVM can support.

Reduce Array Size: Ensure that array sizes are within the JVM limits.

Optimize Data Structures: Consider using more memory-efficient data structures.

Large Stack Frames

Deep recursion or large method calls can consume stack space, which may lead to StackOverflowError, but in some cases, it could contribute to OOM conditions.

RuntimeException

package java.lang;

public class **RuntimeException** extends Exception

public RuntimeException()

public RuntimeException(String message)

public RuntimeException(String message, Throwable cause)

public RuntimeException(Throwable cause)

protected RuntimeException(

String message, 细节信息

Throwable cause, 原因

boolean enableSuppression,

boolean writableStackTrace
)

StackTraceElement

package java.lang;
public final class **StackTraceElement** implements java.io.Serializable　　异常追踪元素

StackWalker

package java.lang;
public final class **StackWalker**
　　The initially called function will be located at the bottom of StackFrames
　　- outside2
　　- outside1
　　- main
public <T> T walk(Function<? super Stream<StackFrame>, ? extends T> function)　　Applies the given function to the stream of StackFrames for the current thread.
　　Walk 内的函数必须返回数值而不是 stream，确保最终 stream 是关闭状态。

IllegalArgumentException

package java.lang;
public class **IllegalArgumentException** extends RuntimeException　非法参数异常

public IllegalArgumentException(String s)　　手动抛出异常

IllegalStateException

package java.lang;
public class **IllegalStateException** extends RuntimeException　　非法状态异常

(System)

Runtime

package java.lang;
public class **Runtime**　　运行信息

public static Runtime getRuntime()
public native int availableProcessors();　返回 JVM 可用处理器的数量

System

package java.lang;
public final class **System**　　系统信息

public final static PrintStream err = null;　　“标准”错误输出流。 此流已打开并准备接受输出数据
　　此输出流用于显示应立即引起用户注意的错误消息或其他信息，即使主要输出流（变量 out 的值）已重定向到文件或其他目标，即通常不会持续监控

public static native long currentTimeMillis();　获取当前毫秒数
public static String getProperty(String key)　获取系统属性
　　line.separator 系统换行符（File.separator 路径分隔符）

user.dir 项目根路径 // System.getProperty("user.dir") = D:\Desktop\DevProject\saidake-manage-project
user.home // C:\Users\saidake
java.io.tmpdir 系统临时目录
FileSystemView.getFileSystemView().getHomeDirectory().getPath(); Desktop directory

public static String lineSeparator() 获取系统换行符
public static native void arraycopy(Object src, int srcPos, Object dest, int destPos, int length) 拷贝一个数组
public static native long nanoTime(); 从某个固定但任意的起始时间开始的纳秒数

System.out.println() 换行输出
System.out.print() 不换行输出
System.out.printf("%d%c%f%s", a, b, c) 格式化输出（int 型 char 型 浮点型 字符串型）

ClassLoader

package java.lang;
public abstract class **ClassLoader** 类加载器
public Enumeration<URL> getResources(String name) throws IOException 获取资源文件的 URL

(Security)

SecurityManager

package java.lang;
public class **SecurityManager** 安全管理 允许应用程序实现安全策略的类。 它允许应用程序在执行可能不安全或敏感的操作之前，确定操作是什么以及是否在允许执行操作的安全上下文中尝试操作。 应用程序可以允许或禁止操作。

权限属于以下类别：文件，套接字，网络，安全性，运行时，属性，AWT，反射和可序列化。 管理这些不同的权限类别类是
java.io.FilePermission，java.net.SocketPermission，
java.net.NetPermission， java.security.SecurityPermission， java.lang.RuntimePermission，
java.util.PropertyPermission ， java.awt.AWTPermission ， java.lang.reflect.ReflectPermission ，
java.io.SerializablePermission
除前两个之外的所有文件（FilePermission 和 SocketPermission）都是 java.security.BasicPermission 子类

public void checkPermission(Permission perm) 如果根据当前有效的安全策略不允许由给定权限指定的请求访问，则抛出 SecurityException 。
线程检查：始终在当前正在执行的线程的上下文中执行安全检查，确定调用线程具有执行所请求的操作的权限。
public void checkPermission(Permission perm, Object context) 除了权限之外还采用上下文对象的 checkPermission 方法，基于该上下文而不是当前执行线程的方式做出访问决策。
public void checkAccess(Thread t) 如果不允许调用线程修改线程参数，则抛出 SecurityException 。
public void checkWrite(String file) 如果不允许调用线程写入字符串参数指定的文件，则抛出 SecurityException 。
public void checkDelete(String file) 如果不允许调用线程删除指定文件，则抛出 SecurityException 。

(annotation)

SafeVarargs

package java.lang;
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.CONSTRUCTOR, ElementType.METHOD})
public @interface **SafeVarargs**

Suppress warnings related to unsafe operations when dealing with varargs (variable-length argument lists).
It applies to methods or constructors that use varargs, and its primary purpose is to ensure that these methods don't expose heap pollution risks, which can occur when generic varargs are passed.

```
@SafeVarargs
public static <T> void safeMethod(T... args) {
    // Safe operations with varargs
}
```

Heap pollution

The term "heap pollution" refers to a situation where the type safety of objects stored in the heap memory is compromised due to improper use of generics.

it refers to corrupting the type system by allowing objects of an unexpected type to be stored in a structure that expects a specific type.

```
public static <T> void method(T... args) {
    Object[] array = args; // Array of type Object[]
    array[0] = 42; // Unsafe assignment: could cause heap pollution if T is not Integer
}
```

Others

package java.lang;

@Override 保证编译时候 Override 函数声明正确性【METHOD】

@SafeVarargs 断定声明的构造函数和方法的主体不会对其 varargs 参数执行潜在的不安全的操作（对于非 static 或非 final 声明的方法，不适用，会编译不通过） 【CONSTRUCTOR, METHOD】

@Deprecated 对不应该再使用的方法添加注解，当编程人员使用这些方法时，会获取提示【CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE】

@SuppressWarnings 关闭特定的警告信息【TYPE, FIELD, METHOD, PARAMETER, CONSTRUCTOR, LOCAL_VARIABLE】

value= “unchcked”

- unchecked 告诉编译器忽略 unchecked 警告信息，如使用 List，ArrayList 等未进行参数化产生的警告信息，执行了未检查的转换时的警告 // unchecked call .....
- deprecation 如果使用了使用@Deprecated 注释的方法，编译器将出现警告信息。使用这个注释将警告信息去掉。
- fallthrough 当 Switch 程序块直接通往下一种情况而没有 break 时的警告
- path 在类路径，源文件路径等中有不存在的路径时的警告
- serial 忽略在 serializable 类中没有声明 serialVersionUID 变量
- serialVersionUID 定义时的警告
- finally 任何 finally 子句不能正常完成时的警告
- all 关于以上所有情况的警告
- rawtypes 抑制没有传递带有泛型的参数的警告
- JavadocReference

```
@SuppressWarnings("unchecked")
  public static <T> T getBean(String name) {    return (T) applicationContext.getBean(name);      }
```

注解处理器

概念： 如果没有用来读取注解的方法和工作，那么注解也就不会比注释更有用处了。使用注解的过程中， 很重要的一部分就是创建于使用注解处理器。

Java SE5 扩展了反射机制的 API，以帮助程序员快速 的构造自定义注解处理器。下面实现一个注解处理器。

FruitProvider >>

```
@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Documented
```

```
public @interface FruitProvider {        // 定义注解
  public int id() default -1;             // 供应商编号
  public String name() default "";        // 供应商名称
  public String address() default "";      //供应商地址
}
```

Apple >>

```
public class Apple {
   @FruitProvider(id = 1, name = "陕西红富士集团", address = "陕西省西安市延安路")      //注解使用
   private String appleProvider;

   public void setAppleProvider(String appleProvider) {
     this.appleProvider = appleProvider;
   }
   public String getAppleProvider() {
     return appleProvider;
   }
}
```

FruitInfoUtil >>

```
public class FruitInfoUtil {                        //注解处理器
   public static void getFruitInfo(Class<?> clazz) {
      String strFruitProvicer = "供应商信息：";
      Field[] fields = clazz.getDeclaredFields();      //通过反射获取处理注解
      for (Field field : fields) {
         if (field.isAnnotationPresent(FruitProvider.class)) {
            FruitProvider fruitProvider = (FruitProvider) field.getAnnotation(FruitProvider.class);
            strFruitProvicer = " 供应商编号：" + fruitProvider.id() + " 供应商名称：" + fruitProvider.name() + " 供应商地址："+
fruitProvider.address();  //注解信息的处理地方
            System.out.println(strFruitProvicer);
         }
      }
   }
}
```

FruitRun >>

```
public class FruitRun {
   public static void main(String[] args) {
      FruitInfoUtil.getFruitInfo(Apple.class);  输出结果：  供应商编号：1 供应商名称：陕西红富士集团 供应商地址：陕西省西安市延
   }
}
```

annotation

Annotation

package java.lang.annotation;
public interface **Annotation** 注解

Class<? extends Annotation> annotationType() 注解类型

RetentionPolicy

```
package java.lang.annotation;
public enum RetentionPolicy {        定义被它所注解的注解保留多久
    SOURCE,          只在源代码级别保留，编译时就会被忽略，在 class 字节码文件中不包含。
    CLASS,           编译时被保留，默认的保留策略，在 class 文件中存在，但 JVM 将会忽略，运行时无法获得。
    RUNTIME          将被 JVM 保留，所以他们能在运行时被 JVM 或其他使用反射机制的代码所读取和使用。
}
```

ElementType

```
package java.lang.annotation;
public enum ElementType {
    TYPE,              接口，类，枚举，注解
    FIELD,             字段，枚举的常量
    METHOD,            方法
    PARAMETER,         方法参数
    CONSTRUCTOR,       构造函数
    LOCAL_VARIABLE,    局部变量
    ANNOTATION_TYPE,   注解
    PACKAGE,           包
    TYPE_PARAMETER,    用来标注类型参数
    TYPE_USE           能标注任何类型名称
}
```

元注解

Annotation：　注解 是 Java 提供的一种对元程序中元素关联信息和元数据（metadata）的途径 和方法。

　　Annatation(注解)是一个接口，程序可以通过反射来获取指定程序中元素的 Annotation 对象，然后通过该 Annotation 对象来获取注解中的元数据信息

元注解：元注解的作用是负责注解其他注解。 Java5.0 定义了 4 个标准的 meta-annotation 类型，它们被 用来提供对其它 annotation 类型作说明。

```
package java.lang.annotation;
@Target    说明了 注解 所修饰的对象范围【ANNOTATION_TYPE】
           注解 可被用于 packages、types（类、 接口、枚举、Annotation 类型）、类型成员（方法、构造方法、成员变量、枚举值）、方法参数 和本地变量（如循环变量、catch 参数）。
           在 注解 类型的声明中使用了 target 可更加明晰 其修饰的目标
      value= “FIELD”
                CONSTRUCTOR      构造器声明
                FIELD            域声明（包括 enum 实例）
                LOCAL_VARIABLE   局部变量声明
                METHOD     方法声明
                PACKAGE    包声明
                PARAMETER  参数声明
                TYPE       类，接口（包括注解类型）或 enum 声明
@Retention     定义了该 注解 被保留的时间长短：表示需要在什么级别保存注解信息，用于描 述注解的生命周期（即：被描述的注解在什么范围内有效）【ANNOTATION_TYPE】
      value= “SOURCE”
                SOURCE   在源文件中有效（即源文件保留）
                CLASS    在 class 文件中有效（即 class 保留）
                RUNTIME  在运行时有效（即运行时保留）
```

@Documented 描述-javadoc。用于描述其它类型的 annotation 应该被作为被标注的程序成员的公共 API，因 此可以被例如 javadoc 此类的工具文档化。【ANNOTATION_TYPE】

@Inherited 阐述了某个被标注的类型是被继承的。如果一 个使用了@Inherited 修饰的 注解 类型被用于一个 class，则这个 注解 将被用于该 class 的子类。【ANNOTATION_TYPE】

reflect

AccessibleObject

package java.lang.reflect;

public class AccessibleObject implements AnnotatedElement 对象访问控制

public void setAccessible(boolean var1) throws SecurityException 取消访问检查（public，private）

Field

package java.lang.reflect;

public final class Field 成员变量映射

extends AccessibleObject implements Member

public String getName() 获取字段名

public Type getGenericType() 获取字段类型

public Object get(Object obj) throws IllegalArgumentException, IllegalAccessException 获取实例的 成员变量值

public void set(Object obj, Object value) throws IllegalArgumentException, IllegalAccessException 设置实例的 成员变量值

Method

package java.lang.reflect;

public final class Method extends Executable 成员方法映射

public Object invoke(Object obj, Object... args) 运行实例的成员方法

public <T extends Annotation> T getAnnotation(Class<T> annotationClass) 获取方法上的注解对象，内部也包含注解的参数成员

Type

package java.lang.reflect;

public interface Type 类型

default String getTypeName() 获取类型名

ParameterizedType

package java.lang.reflect;

public interface ParameterizedType extends Type

Type[] getActualTypeArguments() 此类型实际类型参数的 Type 对象的数组（返回的 Type 强转为 class<?>可以获取泛型的 class）

Array

package java.lang.reflect;

public final class **Array** 提供静态方法创建数组

public static Object newInstance(Class<?> componentType, int length) 创建一个原始数组 // T[] joinedArray = (Object[]) ((Object[]) Array.newInstance(type1, array1.length + array2.length))

Proxy

package java.lang.reflect;

public class **Proxy** implements java.io.Serializable 代理对象，会通过 InvocationHandler 参数执行 invoke 方法。

Save class files generated by JDK dynamic proxy in local:

System.getProperties().put(""sun.misc.ProxyGenerator.saveGeneratedFiles", "true");

System.getProperties().put("jdk.proxy.ProxyGenerator.saveGeneratedFiles", "true");

D:\Desktop\DevProject\saidake-manage-project\jdk\proxy1\$Proxy0.class

public static Object newProxyInstance(ClassLoader loader, Class<?>[] interfaces, InvocationHandler h)

public static boolean isProxyClass(Class<?> cl) 是否是代理类

Define the Interface

The interface should define the methods that the proxy will delegate to the actual implementation.

```
public interface MyService {
    void performAction();
}
```

Create the Real Implementation

Implement the interface with the actual business logic.

```
public class MyServiceImpl implements MyService {
    @Override
    public void performAction() {
        System.out.println("Performing action...");
    }
}
```

Implement the InvocationHandler

The InvocationHandler interface is used to handle method calls on the proxy instance. It provides a way to intercept and process method calls.

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;

public class MyInvocationHandler implements InvocationHandler {
    private final Object target;

    public MyInvocationHandler(Object target) {
        this.target = target;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("Before method: " + method.getName());
        Object result = method.invoke(target, args);
        System.out.println("After method: " + method.getName());
        return result;
    }
}
```

Create the Proxy Instance

Use Proxy.newProxyInstance to create the proxy instance. This method takes the class loader, an array of interfaces, and the InvocationHandler.

```
import java.lang.reflect.Proxy;

public class ProxyExample {
    public static void main(String[] args) {
        MyService realService = new MyServiceImpl();

        MyService proxyInstance = (MyService) Proxy.newProxyInstance(
            MyService.class.getClassLoader(),
            new Class<?>[]{MyService.class},
            new MyInvocationHandler(realService)
        );
```

```
        proxyInstance.performAction();
    }
}
```

ProxyGenerator

package java.lang.reflect;
final class **ProxyGenerator** extends ClassWriter
private static final boolean saveGeneratedFiles = java.security.AccessController.doPrivileged(new GetBooleanAction("jdk.proxy.ProxyGenerator.saveGeneratedFiles"));
static byte[] generateProxyClass(ClassLoader loader, final String name, List<Class<?>> interfaces, int accessFlags)

InvocationHandler

package java.lang.reflect;
public interface **InvocationHandler**
public Object invoke(Object proxy, Method method, Object[] args) 执行目标类中所有的方法都会经过 invoke 方法

InvocationTargetException

package java.lang.reflect;
public class **InvocationTargetException** extends ReflectiveOperationException

java.math

BigDecimal

package java.math;
public class **BigDecimal** extends Number implements Comparable<BigDecimal> 如果需要精确计算，必须用 String 来够造 BigDecimal，Java 里面的商业计算，不能用 float 和 double，因为他们无法 进行精确计算

percision 总位数
scale 保留小数

public BigDecimal(String val) 传入字符串，null，"" 都会报错 // new BigDecimal("2.225667")
public BigDecimal(int val)
public BigDecimal(double val) 这种写法不允许，会造成精度损失

public BigDecimal add(BigDecimal augend) 被加数
public BigDecimal setScale(int newScale, int roundingMode) 保留位数，多余位数处理方式

public BigDecimal multiply(BigDecimal m_val); 乘法
public BigDecimal multiply(BigDecimal m_val, MathContext ma_co); 乘法

public final static int ROUND_UP = 0; 进位处理 // 2.225667 ==> 2.23
public final static int ROUND_DOWN = 1; 直接去掉多余的位数 // 2.225667 ==> 2.22
public final static int ROUND_CEILING = 2; 向上 // 2.225667 ==> 2.23 -2.225667 ==> -2.22
public final static int ROUND_FLOOR = 3; 向下 // 2.225667 ==> 2.22 -2.225667 ==> -2.23
public final static int ROUND_HALF_UP = 4; 四舍五入（若舍弃部分>=.5，就进位） // 2.225667 ==> 2.23
public final static int ROUND_HALF_DOWN = 5; 四舍五入（若舍弃部分>.5,就进位） // 2.225667 ==> 2.22
public final static int ROUND_HALF_EVEN = 6; 如果舍弃部分左边的数字为偶数 则向下，如果舍弃部分左边的数字为奇数 则向上
// 4.05 ==> 4.0 4.15 ==> 4.2

public final static int ROUND_UNNECESSARY = 7; 断言请求的操作具有精确的结果，因此不需要舍入

RoundingMode

BigInteger

package java.math;

public class **BigInteger** extends Number implements Comparable<BigInteger> 表示任意大小的整数。 BigInteger 内部用一个 int[] 数组来模拟一个非常大的整数

public static final BigInteger TEN = valueOf(10) 表示 10

java.net

URI

package java.net;

public final class **URI** implements Comparable<URI>, Serializable 统一资源定位符

URL

package java.net;

public final class **URL** implements java.io.Serializable 类 URL 表示统一资源定位符，指向万维网上的“资源”的指针

public URL(String spec) throws MalformedURLException 解析一个字符串网址

public URL(
    String protocol, 协议
    String host, 主机地址
    int port, 端口
    String file) 主机上的文件名
    throws MalformedURLException

InetAddress

package java.net;

public class **InetAddress** implements java.io.Serializable 表示 ip 地址

public static InetAddress getByName(String host) 确定主机名称的 IP 地址。主机名称可以是机器名称，也可以是 IP 地址

public static InetAddress getLocalHost() throws UnknownHostException 根据网卡取本机内网 IP // InetAddress.getLocalHost().getHostAddress();

public byte[] getAddress() 返回文本显示中的 IP 地址字符串

public String getHostName() 获取此 IP 地址的主机名

DatagramSocket

package java.net;

public class **DatagramSocket** implements java.io.Closeable 单播广播客户端/服务端

public DatagramSocket() throws SocketException 空参绑定随机端口

public DatagramSocket(int port) throws SocketException 绑定指定端口

public void send(DatagramPacket p) throws IOException 发送一个数据包

public synchronized void receive(DatagramPacket p) throws IOException 接收一个数据包

public void close() 关闭连接

Client >>

```
// 找码头
DatagramSocket ds = new DatagramSocket();
// 打包礼物
String s ="送给村长老丈人的礼物";
byte[] bytes = s.getBytes();
InetAddress address = InetAddress.getByName("127.0.0.1");
int port = 1000o;
DatagramPacket dp = new DatagramPacket(bytes,bytes.length, address,port);
//由码头发送包裹
ds.send(dp);
//付钱走羊
ds.close();
```

Server >>

```
// 找码头----表示接收端从 10000 塔口接收致据的.
DatagramSocket ds = new DatagramSocket( 10080);
// 创建一个新的箱子
byte []bytes = new byte[1024];
DatagramPacket dp = new DatagramPacket(bytes,bytes.length);
// 接收礼物,把礼物放到新的箱子中
ds.receive(dp);
// 从新的箱子里面获取礼物
byte[]data = dp.getData();
System.out.println(new String(data));
// 拿完走羊
ds.close();
```

DatagramPacket

package java.net;

public final class **DatagramPacket**　数据包

public DatagramPacket(byte buf[], int offset, int length, InetAddress address, int port)　构造一个数据包，发送长度为 length 的数据包到指定主机上的指定端口号。// InetAddress.getByName("255.255.255.255"); 广播，路由器看到这个地址就会发送给所有主机

public DatagramPacket(byte buf[], int length)　构造一个 DatagramPacket 用于接收长度为 length 数据包。

MulticastSocket

package java.net;

public class **MulticastSocket** extends DatagramSocket　组播客户端/服务端

public void joinGroup(InetAddress mcastaddr) throws IOException　把当前计算机绑定一个组播地址,表示添加到这一组中. // InetAddress.getByName("224.0.1.0")

ConnectException

package java.net;

public class **ConnectException** extends SocketException　连接异常

HttpURLConnection

package java.net;
abstract public class **HttpURLConnection** extends URLConnection　　Http 连接

```
connection.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");              设置请求头
connection.setRequestProperty("Content-Length", String.valueOf(param.getBytes().length));

connection.setDoOutput(true);        允许向服务器写入写出数据
 connection.setDoInput(true);

OutputStream outputStream = connection.getOutputStream();        获取输出流，向服务器写入数据。
outputStream.write(param.getBytes());
outputStream.flush();
outputStream.close();


int statusCode = connection.getResponseCode();  获取响应码，判断请求是否成功。

InputStream inputStream = statusCode == 200 ? connection.getInputStream() : connection.getErrorStream();
     读取响应数据。
BufferedReader reader = new BufferedReader(new InputStreamReader(inputStream));
StringBuilder response = new StringBuilder();
String line;
while ((line = reader.readLine()) != null) {
   response.append(line);
}
reader.close();
inputStream.close();
```

protected HttpURLConnection (URL u)

ServerSocket

package java.net;
public class **ServerSocket** implements java.io.Closeable

public Socket accept() throws IOException
Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made.

A new Socket s is created and, if there is a security manager,
the security manager's checkAccept method is called with s.getInetAddress().getHostAddress() and s.getPort() as its arguments to ensure the operation is allowed.
This could result in a SecurityException.

URLDecoder

package java.net;
public class **URLDecoder**　　urlj 解码器
public static String decode(String s, String enc) throws UnsupportedEncodingException　　用编码解码一个字符串　　// URLDecoder.decode(keyFileName, "UTF-8"))

Buffer

package java.nio;

public abstract class **Buffer**

A Buffer is a container for a fixed amount of data of a specific primitive type.

Buffers are used for reading and writing data to and from NIO channels.

Child Classes: ByteBuffer、IntBuffer、 CharBuffer、 LongBuffer、 DoubleBuffer、FloatBuffer、 ShortBuffer

Example:

```
import java.nio.ByteBuffer;

public class BufferExample {
    public static void main(String[] args) {
        // Create a ByteBuffer with a capacity of 10 bytes
        ByteBuffer buffer = ByteBuffer.allocate(10);

        // Write data into the buffer. The position is updated with each write.
        buffer.put((byte) 1);
        buffer.put((byte) 2);
        buffer.put((byte) 3);

        // The buffer is flipped to prepare it for reading.
       // The limit is set to the current position, and the position is reset to zero.
        buffer.flip();

        // Read data from the buffer
        // Data is read from the buffer using the get method until there are no more elements
remaining
        // (i.e., position reaches limit).
        while (buffer.hasRemaining()) {
            System.out.println(buffer.get());
        }
    }
}
```

public final Buffer limit(int newLimit)　　A container for data of a specific primitive type.

public Buffer flip()

Flips this buffer. The limit is set to the current position and then the position is set to zero. If the mark is defined then it is discarded.

ByteBuffer

package java.nio;

public abstract class **ByteBuffer** extends Buffer implements Comparable<ByteBuffer>　　A byte buffer, This class defines six categories of operations upon byte buffers

public static ByteBuffer wrap(

byte[] array,　　将备份新缓冲区的数组

int offset,　　子数组偏移量，必须为非负数

int length　　子数组长度

)

public static ByteBuffer allocate(int capacity)　　Allocates a new byte buffer.

channels

Selector

package java.nio.channels;

public abstract class **Selector** implements Closeable

A Selector allows a single thread to manage multiple SelectableChannel objects (such as SocketChannel and ServerSocketChannel),
making it possible to handle multiple connections concurrently.

Key Components of Selector:

- SelectableChannel Represents a channel that can be multiplexed with a selector.
- SelectionKey Represents the relationship between a selectable channel and a selector.
  It contains information about the channel, the selector, and the interest set (operations of interest, such as read or write).
- Interest Set The set of operations the channel is interested in (e.g., SelectionKey.OP_READ, SelectionKey.OP_WRITE).
- Ready Set The set of operations the channel is ready for.
- Select The method used to block until at least one channel is ready for the operations specified in its interest set.

Example:

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.*;
import java.net.InetSocketAddress;
import java.util.Iterator;
import java.util.Set;

public class SelectorExample {
    public static void main(String[] args) throws IOException {
        // Create a selector
        Selector selector = Selector.open();

        // Create a non-blocking server socket channel
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.configureBlocking(false);
        serverSocketChannel.socket().bind(new InetSocketAddress(8080));

        // Register the server socket channel with the selector for accept events
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

        while (true) {
            // Wait for an event
            selector.select();

            // Get the set of selection keys
            Set<SelectionKey> selectedKeys = selector.selectedKeys();
            Iterator<SelectionKey> iterator = selectedKeys.iterator();

            while (iterator.hasNext()) {
                SelectionKey key = iterator.next();
                iterator.remove();

                if (key.isAcceptable()) {
                    // Accept the new connection
                    ServerSocketChannel serverChannel = (ServerSocketChannel) key.channel();
                    SocketChannel socketChannel = serverChannel.accept();
                    socketChannel.configureBlocking(false);

                    // Register the new connection with the selector for read events
                    socketChannel.register(selector, SelectionKey.OP_READ);
                } else if (key.isReadable()) {
                    // Read data from the connection
                    SocketChannel socketChannel = (SocketChannel) key.channel();
                    ByteBuffer buffer = ByteBuffer.allocate(256);
                    socketChannel.read(buffer);

                    String result = new String(buffer.array()).trim();
```

```
                        System.out.println("Message received: " + result);

                        // Close the connection
                        socketChannel.close();
                    }
                }
            }
        }
    }
```

public static Selector open() throws IOException

Opens a selector.

The new selector is created by invoking the openSelector method of the system-wide default SelectorProvider object.

public abstract Set<SelectionKey> keys(); Returns this selector's key set.

public abstract Set<SelectionKey> selectedKeys(); Returns this selector's selected-key set.

public abstract int select() throws IOException;

Selects a set of keys whose corresponding channels are ready for I/O operations.

This method performs a blocking selection operation. It returns only after at least one channel is selected,

this selector's wakeup method is invoked, or the current thread is interrupted, whichever comes first.

public abstract int select(long timeout) throws IOException;

Selects a set of keys whose corresponding channels are ready for I/O operations.

This method performs a blocking selection operation. It returns only after at least one channel is selected,

this selector's wakeup method is invoked, the current thread is interrupted, or the given timeout period expires, whichever comes first.

public abstract int selectNow() throws IOException;

Selects a set of keys whose corresponding channels are ready for I/O operations.

This method performs a non-blocking selection operation. If no channels have become selectable since the previous selection operation then this method immediately returns zero.

public abstract Selector wakeup();

Causes the first selection operation that has not yet returned to return immediately.

If another thread is currently blocked in a selection operation then that invocation will return immediately.

If no selection operation is currently in progress then the next invocation of a selection operation will return immediately unless selectNow() or selectNow(Consumer) is invoked in the meantime.

In any case the value returned by that invocation may be non-zero. Subsequent selection operations will block as usual unless this method is invoked again in the meantime.

Channel

package java.nio.channels;

public interface **Channel** extends Closeable

Channels and Streams in I/O are on similar levels.

Streams are unidirectional (e.g., InputStream, OutputStream), whereas Channels are bidirectional, meaning they can be used for both read and write operations.

In NIO (New I/O), the main implementations of Channels are:

- **FileChannel**: File I/O
- **DatagramChannel**: UDP
- **SocketChannel**: TCP Client
- **ServerSocketChannel**: TCP Server

SocketChannel

package java.nio.channels;
public abstract class **SocketChannel**
  extends AbstractSelectableChannel
  implements ByteChannel, ScatteringByteChannel, GatheringByteChannel, NetworkChannel

public abstract int read(ByteBuffer dst) throws IOException;
public abstract int write(ByteBuffer src) throws IOException;

SelectableChannel

package java.nio.channels;
public abstract class **SelectableChannel**
  extends AbstractInterruptibleChannel
  implements Channel

ServerSocketChannel

package java.nio.channels;
public abstract class **ServerSocketChannel**
  extends AbstractSelectableChannel
  implements NetworkChannel

public static ServerSocketChannel open() throws IOException
    Opens a server-socket channel for an Internet protocol socket.

public abstract SocketChannel accept() throws IOException;
    Accepts a connection made to this channel's socket.
    If this channel is in non-blocking mode then this method will immediately return null if there are no pending connections.
    Otherwise it will block indefinitely until a new connection is available or an I/O error occurs.

public final ServerSocketChannel bind(SocketAddress local) throws IOException
    Binds the channel's socket to a local address and configures the socket to listen for connections.
    An invocation of this method is equivalent to the following:
```
bind(local, 0);
```

spi

AbstractSelectableChannel

package java.nio.channels.spi;
public abstract class **AbstractSelectableChannel**
      extends SelectableChannel

public final SelectableChannel configureBlocking(boolean block) throws IOException
          Adjusts this channel's blocking mode.

charset

Charset

package java.nio.charset;
public abstract class **Charset** 字符集

public static Charset defaultCharset()　　返回 java 虚拟机的默认字符集
public static Charset forName(String charsetName)　根据名称获取字符集
public final String name()　获取当前字符集的经典名称

StandardCharsets

package java.nio.charset;
public final class **StandardCharsets**　标准字符集
public static final Charset US_ASCII = Charset.forName("US-ASCII");
public static final Charset ISO_8859_1 = Charset.forName("ISO-8859-1");
public static final Charset UTF_8 = Charset.forName("UTF-8");
public static final Charset UTF_16BE = Charset.forName("UTF-16BE");
public static final Charset UTF_16LE = Charset.forName("UTF-16LE");
public static final Charset UTF_16 = Charset.forName("UTF-16");　　Sixteen-bit UCS Transformation Format, byte order identified by an optional byte-order mark

file

Files

package java.nio.file;
public final class **Files**　文件工具
　　java.nio.file.Paths　路径工具

public static Stream<Path> walk(Path start, FileVisitOption... options) throws IOException　遍历文件夹（包含子文件夹及子其文件），遍历结果是一个 Stream

public static Path copy(Path source, Path target, CopyOption... options) throws IOException
public static List<String> readAllLines(Path path, Charset cs) throws IOException
public static Stream<String> lines(Path path) throws IOException
　This method reads empty lines as well.
　`//Files.lines("example.txt").skip(8)　　Skip the first 8 lines`
public static byte[] readAllBytes(Path path) throws IOException
public static String readString(Path path) throws IOException
　　Path filePath = Paths.get("C:/", "temp", "test.txt");
　　String content = Files.readString(filePath);
public static Path writeString(
　　Path path,　　写入路径
　　CharSequence csq,　　写入字符串
　　Charset cs,　　编码
　　OpenOption... options)
　　throws IOException

　　// Files.writeString(Path.of(accountFile.toString()),fileContent, StandardOpenOption.WRITE);　StandardOpenOption will prevent writing to this file.
public static Path writeString(
　　Path path,
　　CharSequence csq,
　　OpenOption... options)
　　throws IOException

// Files.writeString(Path.of(accountFile.toString()),fileContent, StandardOpenOption.WRITE);
public static Path walkFileTree(Recursively traverses the directory tree.
Path start,
Set<FileVisitOption> options,
int maxDepth,
FileVisitor<? super Path> visitor)
public static Path createTempFile(
String prefix,
String suffix,
FileAttribute<?>... attrs
) throws IOException

OpenOption

package java.nio.file;
public interface **OpenOption**

StandardOpenOption

package java.nio.file;
public enum **StandardOpenOption** implements OpenOption 标准打开方式
READ, 标准读
WRITE, 标准写
APPEND, WRITE 写入时，追加
TRUNCATE_EXISTING, WRITE 写入时，文件长度会被截断为 0
CREATE, 不存在就创建，CREATE_NEW 存在时此选项将会被忽略
CREATE_NEW, 创建新文件
DELETE_ON_CLOSE, 关闭时删除
SPARSE, CREATE_NEW 创建新文件时，创建稀疏文件
SYNC, 文件内容和元数据同步写入
DSYNC; 文件内容同步写入

Paths

package java.nio.file;
public final class **Paths** 路径工具
public static Path get(String first, String... more) Splicing Path // Paths.get("D:\\Desktop\\DevProject\\saidake-manage-project\\smp-service\\smp-oracle\\src\\test\\java\\com\\saidake\\target.properties");
public static Path get(URI uri) Obtain path based on URI

Path

package java.nio.file;
public interface **Path** extends Comparable<Path>, Iterable<Path>, Watchable
public static Path of(String first, String... more)
Create a Path
Path.of() automatically handles platform-specific separators (\ for Windows, / for Unix-like systems).
private static final String TEMP_FOLDER = "C:\\Users\\simi\\Desktop\\DevProjects\\simi-docs";
private static final Path DOC_KEY_POINTS_PATH = Path.of(TEMP_FOLDER, "temp", "doc-key-points.txt");
public static Path of(URI uri)
Create a Path based on the resource path.
Path getRoot();
Root Path // D:\
Path getFileName();

File Name // test.txt
String toString();
Output path // D:\Desktop\DevProject\saidake-manage-project\sdk-service\sdk-generator\src\test\java\com\saidake\test.txt
Path resolve(Path other);
Join a other path

Watchable

package java.nio.file;
public interface **Watchable**

FileVisitor

package java.nio.file;
public interface **FileVisitor**<T>
FileVisitResult preVisitDirectory(T dir, BasicFileAttributes attrs) throws IOException;
Called before accessing any directory.
FileVisitResult visitFile(T file, BasicFileAttributes attrs) throws IOException;
Called after accessing any directory.
FileVisitResult visitFileFailed(T file, IOException exc) throws IOException;
Called when accessing each file.
FileVisitResult postVisitDirectory(T dir, IOException exc) throws IOException;
Called on file access failure.

FileVisitResult

package java.nio.file;
public enum **FileVisitResult**
CONTINUE,
Continue the traversal.
TERMINATE,
Immediately terminates the traversal.
SKIP_SUBTREE,
Continue without visiting the entries in this directory. This result is only meaningful when returned from the preVisitDirectory method;
Otherwise this result type is the same as returning CONTINUE.
SKIP_SIBLINGS;
Continue without visiting the siblings of this file or directory.
If returned from the preVisitDirectory method then the entries in the directory are also skipped and the postVisitDirectory method is not invoked.

java.security

Security

package java.security;
public final class **Security** 管理提供者

public static int addProvider(Provider provider) 添加提供者

AccessController

package java.security;
public final class **AccessController** 访问控制器

public static native <T> T doPrivileged(PrivilegedAction<T> action);

允许在一个类实例中的代码通知这个 AccessController：它的代码主体是享受"privileged(特权的)

在做访问控制决策时，如果 checkPermission 方法遇到一个通过 doPrivileged 调用而被表示为 "特权"的调用者，并且没有上下文自变量，checkPermission 方法则将终止检查。

如果那个调用者的域具有特定的许可，则不做进一步检查，checkPermission 安静地返回，表示那个访问请求是被允许的；

如果那个域没有特定的许可，则象通常一样，一个异常被抛出。

MessageDigest

package java.security;

public abstract class **MessageDigest** extends MessageDigestSpi

The MessageDigest class in Java is used for computing cryptographic hash functions.

A cryptographic hash function takes an input (or "message") and returns a fixed-size string of bytes, typically a digest that is unique to the input.

Even small changes to the input will produce a significantly different output, making hash functions useful for various security-related applications.

One-Way:

The hash function is one-way, meaning you can't reverse the hash to get the original input.

Fixed-Length Output:

The output (digest) is of fixed length, regardless of the input size. For example, SHA-256 always produces a 256-bit (32-byte) output.

Efficient:

Hash functions are designed to be fast and efficient to compute.

Common Hash Algorithms:

MD2	A 128-bit hash algorithm.
MD5	A 128-bit hash algorithm, widely used but considered cryptographically broken and unsuitable for further use.
SHA-1	A 160-bit hash algorithm, also considered broken and deprecated for most cryptographic uses.
SHA-224	A truncated version of SHA-256, producing a 224-bit hash.
SHA-256	A member of the SHA-2 family, producing a 256-bit hash.
SHA-384	A member of the SHA-2 family, producing a 384-bit hash.
SHA-512	A member of the SHA-2 family, producing a 512-bit hash.
SHA-512/224	A variant of SHA-512, truncated to produce a 224-bit hash.
SHA-512/256	A variant of SHA-512, truncated to produce a 256-bit hash.

javax.crypto.Cipher

public void update(byte input)
public void update(byte[] input)
public void update(byte[] input, int offset, int len)
public final void update(ByteBuffer input)

public Object clone() throws CloneNotSupportedException
public byte[] digest()
public byte[] digest(byte[] input)
public int digest(byte[] buf, int offset, int len) throws DigestException

public final String getAlgorithm()　返回标识算法的独立于实现细节的字符串。
public final int getDigestLength()　返回以字节为单位的摘要长度，如果提供程序不支持此操作并且实现是不可复制的，则返回 0。
public final Provider getProvider()　返回此信息摘要对象的提供程序。
public void reset()　重置摘要以供再次使用。

public static MessageDigest getInstance(String algorithm) throws NoSuchAlgorithmException　生成实现指定摘要算法的 MessageDigest 对象。
public static MessageDigest getInstance(String algorithm, String provider) throws NoSuchAlgorithmException, NoSuchProviderException　生成实现指定提供程序提供的指定算法的 MessageDigest 对象，如果该算法可从指定的提供程序得到的话。
public static boolean isEqual(byte[] digesta, byte[] digestb)　比较两个摘要的相等性。

Usage

```
public static String encryptMode(String str, String mode) throws NoSuchAlgorithmException {
    MessageDigest md = MessageDigest.getInstance(mode);
    md.update(str.getBytes());
    byte[] bt = md.digest();
    StringBuffer buffer = new StringBuffer();
    for (int i = 0; i < bt.length; i++) {
        buffer.append(Character.forDigit((bt[i] & 240) >> 4, 16));      Number
        buffer.append(Character.forDigit(bt[i] & 15, 16));              String
    }
    return buffer.toString();
}
```

SHA-256 algorithm

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;

public class MessageDigestExample {
    public static void main(String[] args) {
        try {
            // Create MessageDigest instance for SHA-256
            MessageDigest md = MessageDigest.getInstance("SHA-256");

            // Provide the input to the MessageDigest
            String input = "Hello, World!";
            byte[] hash = md.digest(input.getBytes());

            // Convert the byte array to a hexadecimal string
            StringBuilder hexString = new StringBuilder();
            for (byte b : hash) {
                hexString.append(String.format("%02x", b));
            }

            System.out.println("SHA-256 Hash: " + hexString.toString());
        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
    }
}
```

KeyStore

package java.security;
public class **KeyStore**

public static KeyStore getInstance(String type)　返回一个指定类型的 KeyStore 对象 【JKS, PKCS12 】
public final static String getDefaultType()　　获取默认 KeyStore 类型 jks
public final void load(InputStream stream, char[] password)　　throws IOException, NoSuchAlgorithmException, CertificateException　从输入流中加载一个证书，即使是新文件证书，也需要调用 // load(null, "xxx")
public final void store(OutputStream stream, char[] password) throws KeyStoreException, IOException, NoSuchAlgorithmException, CertificateException　保存
public final String getType()　获取证书类型
public final Enumeration<String> aliases() throws KeyStoreException　获取证书内的所有 alia
public final boolean isKeyEntry(String alias)　throws KeyStoreException　alia 是否是一个 密钥对
public final Key getKey(String alias, char[] password)　　throws KeyStoreException, NoSuchAlgorithmException, UnrecoverableKeyException　返回和 alia 关联的 key
public final Certificate[] getCertificateChain(String alias) throws KeyStoreException　　返回和 alia 关联的 证书链

public final Certificate getCertificate(String alias) throws KeyStoreException　　返回和 alia 关联的 证书（私钥只有一个）
public final void setKeyEntry(String alias, Key key, char[] password, Certificate[] chain) throws KeyStoreException　设置一个 密钥对

Import Cert

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.security.KeyStore;
import java.security.cert.Certificate;
import java.security.cert.CertificateFactory;

public class ImportCert {
    public static void main(String[] args) throws Exception {
        // Path to your keystore
        String keystorePath = "path/to/your/keystore.jks";
        // Password for the keystore
        char[] keystorePassword = "keystorepassword".toCharArray();

        // Load the existing keystore
        FileInputStream is = new FileInputStream(keystorePath);
        KeyStore keystore = KeyStore.getInstance(KeyStore.getDefaultType());
        keystore.load(is, keystorePassword);
        is.close();

        // Load the certificate you want to import
        FileInputStream certInputStream = new FileInputStream("path/to/your/certificate.crt");
        CertificateFactory cf = CertificateFactory.getInstance("X.509");
        Certificate cert = cf.generateCertificate(certInputStream);
        certInputStream.close();

        // Set the certificate entry in the keystore
        String alias = "myalias";
        keystore.setCertificateEntry(alias, cert);

        // Save the updated keystore
        FileOutputStream fos = new FileOutputStream(keystorePath);
        keystore.store(fos, keystorePassword);
        fos.close();
    }
}
```

Print Cefificate Infomation

```
import java.io.FileInputStream;
import java.security.KeyStore;
import java.security.cert.Certificate;
import java.security.cert.X509Certificate;
```

```java
public class PrintKeystoreCertificates {
    public static void main(String[] args) throws Exception {
        String keystorePath = "path/to/your/keystore.jks";
        String keystorePassword = "your-keystore-password";

        KeyStore keyStore = KeyStore.getInstance("JKS");
        try (FileInputStream fis = new FileInputStream(keystorePath)) {
            keyStore.load(fis, keystorePassword.toCharArray());
        }

        for (String alias : keyStore.aliases()) {
            Certificate cert = keyStore.getCertificate(alias);
            if (cert instanceof X509Certificate) {
                X509Certificate x509Cert = (X509Certificate) cert;
                System.out.println("Alias: " + alias);
                System.out.println("Subject: " + x509Cert.getSubjectX500Principal());
                System.out.println("Issuer: " + x509Cert.getIssuerX500Principal());
                System.out.println("Serial Number: " + x509Cert.getSerialNumber());
                System.out.println("Valid From: " + x509Cert.getNotBefore());
                System.out.println("Valid Until: " + x509Cert.getNotAfter());
                System.out.println("Signature Algorithm: " + x509Cert.getSigAlgName());
                System.out.println();
            }
        }
    }
}
```

Sample Output

```
Alias: mycert
Subject: X500Principal: CN=example.com, OU=IT, O=Example Corp, L=City, ST=State, C=US
Issuer: X500Principal: CN=Example CA, O=Example CA Org, C=US
Serial Number: 1234567890abcdef
Valid From: Wed Jan 01 00:00:00 PST 2023
Valid Until: Fri Jan 01 00:00:00 PST 2026
Signature Algorithm: SHA256withRSA

Alias: anothercert
Subject: X500Principal: CN=another.com, OU=IT, O=Another Corp, L=City, ST=State, C=US
Issuer: X500Principal: CN=Another CA, O=Another CA Org, C=US
Serial Number: abcdef1234567890
Valid From: Tue Feb 01 00:00:00 PST 2023
Valid Until: Thu Feb 01 00:00:00 PST 2026
Signature Algorithm: SHA256withRSA
```

Key

package java.security;

public interface **Key** extends java.io.Serializable 密钥

public byte[] getEncoded() 获取编码内容

读取密钥——

ObjectInputStream ois = new ObjectInputStream(new FileInputStream(path));

Key key = Key.class.cast(ois.readObject());

ois.close();

PrivateKey

package java.security;

public interface **PrivateKey** extends Key, javax.security.auth.Destroyable 私钥

PublicKey

package java.security;
public interface **PublicKey** extends Key　公钥

KeyFactory

public class **KeyFactory**
xpublic static KeyFactory getInstance(String algorithm) throws NoSuchAlgorithmException
The following is an example of how to use a key factory in order to instantiate a DSA public key from its encoding. Assume Alice has received a digital signature from Bob. Bob also sent her his public key (in encoded format) to verify his signature. Alice then performs the following actions:

```
X509EncodedKeySpec bobPubKeySpec = new X509EncodedKeySpec(bobEncodedPubKey);
KeyFactory keyFactory = KeyFactory.getInstance("DSA");
PublicKey bobPubKey = keyFactory.generatePublic(bobPubKeySpec);
Signature sig = Signature.getInstance("DSA");
sig.initVerify(bobPubKey);
sig.update(data);
sig.verify(signature);
```

KeyPair

package java.security;
public final class **KeyPair** implements java.io.Serializable　密钥对，包括公钥和私钥，私钥需要有自己的密钥（PrivateKey）

public KeyPair(PublicKey publicKey, PrivateKey privateKey)

KeyPairGenerator

package java.security;
public abstract class **KeyPairGenerator** extends KeyPairGeneratorSpi　密钥对生成器
public static KeyPairGenerator getInstance(String algorithm, String provider) throws NoSuchAlgorithmException, NoSuchProviderException　获取 实例
algorithm= RSA　算法
provider= SUN　提供者
public static KeyPairGenerator getInstance(String algorithm) throws NoSuchAlgorithmException　根据算法获取实例
public void initialize(int keysize)　初始化　// initialize(2048)
public KeyPair generateKeyPair()　生成生成 根证书公钥与私钥对

Signature

package java.security;
public abstract class **Signature** extends SignatureSpi 签名
public static Signature getInstance(String algorithm) throws NoSuchAlgorithmException 获取一个实现了签名算法的签名
public final void initVerify(Certificate certificate) throws InvalidKeyException　初始化用于验证的对象，使用证书的公钥
public final void update(byte[] data) throws SignatureException　更新证书信息，用于验证
public final boolean verify(byte[] signature) throws SignatureException　验证证书的签名

SecureRandom

package java.security;
public class **SecureRandom** extends java.util.Random
public static SecureRandom getInstance(String algorithm) throws NoSuchAlgorithmException　生成指定算法的 安全随机生成器
// SecureRandom.getInstance("SHA1PRNG");

synchronized public void setSeed(byte[] seed)　重新设置此随机对象的种子。给定的种子补充而不是替换现有种子。因此，保证重复调用永远不会降低随机性。

Certificate

package java.security.cert;

public abstract class **Certificate** implements java.io.Serializable　证书

数字证书有多种文件编码格式，主要包含 CER 编码、DER 编码等：

CER(Canonical Encoding Rules，规范编码格式)：　是数字证书的一种编码格式，它是 BER(Basic Encoding Rules，基本编码格式)的一个变种，比 BER 规定得更加严格。

DER(Distinguished Encoding Rule, 卓越编码格式)：　同样是 BER 的一个变种，与 CER 不的同之处在于：DER 使用定长模式，而 CER 使用变长模式。

PKCS(Public-Key Cryptography Standards，公钥加密标准)：　由 RSA 实验室和其它安全系统开发商为促进公钥密码发展而制定的一系列标准。

其中 CER、DER 格式证书都符合公钥基础设施（PKI）制定的 X509 国际标准（X.509 标准），统称为 X509 格式证书。

数字证书中包含的信息：版本号、序列号、签名算法、签名哈希算法、颁发者 ISSUER、有效期、使用者 SUBJECT、公钥、指纹、指纹算法，扩展属性信息。

DN：Distinguish Name　颁发者或使用者的 标识名称

格式：CN=姓名，OU=组织单位名称，O=组织名称，L=城市或区域名称，ST=省/市/自治区名称，C=国家双字母

浏览器验证时，CN 与域名完全一致。

指纹：指纹是一个证书的签名，是通过指纹算法 sha1 计算出来的一个 hash 值，用于验证证书内容是否有被篡改。

证书在发布前，CA 机构会把所颁发证书的内容通过指纹算法计算得到一个 hash 值，并用自己的根私钥加密，得到一个签名，　这个加密 hash 值只有对应的公钥才能解密

所以在验证证书时，使用同样的指纹算法将证书内容计算得到一个 hash 值，与公钥解密的 hash 进行比对，就代表证书没有被篡改过。

证书颁发：其实就是使用证书颁发者的私钥对证书使用者的证书进行签名，并设置使用者证书的颁发者，证书一般情况下需要由权威的证书认证机构颁发，其原因就是对证书进行签名使用的是私钥，私钥只有颁发机构才有

证书验证：获取证书颁发机构，从系统中寻找此颁发机构是否为信任机构。

指纹验证证书是否被篡改

CA (Certificate Authority)：GeoTrust，

根证书：CA 机构除了给别人颁发证书外，它自己也有证书，也就是根证书。

根证书也有自己的公钥和私钥，称为根公钥和根私钥，根公钥加密算法是向外公布的，根私钥是保密的

自签名证书：使用的数字签名是网站自己的私钥，而不是来自 CA

使用自签名证书，没有外部权威机构验证服务器是否是它声称的身份，浏览器认为自签名证书不可信。

.p12、.pfx　PKCS12 为个人信息交换语法标准，故个人信息证书采用该格式

.p10、.csr　PKCS10 为证书请求语法标准，故证书请求文件采用该格式

.p7b、.p7c、.spc　PKCS7 为密码消息语法标准

public abstract PublicKey getPublicKey()　获取证书的公钥

public abstract byte[] getEncoded() throws CertificateEncodingException　返回此证书的编码结果　// FileOutputStream fos = new FileOutputStream("H:/certtest/ca.cer ");　fos.write(encoded);　fos.close()

X509Certificate

package java.security.cert;

public abstract class **X509Certificate** extends Certificate implements X509Extension　X.509 证书，提供了一个标准访问 X.509 证书的所有属性的方法

public abstract String getSigAlgName()　获取证书的签名算法　// SHA256withRSA

public abstract byte[] getTBSCertificate() throws CertificateEncodingException 获取 DER 编码的证书信息
public abstract byte[] getSignature() 获取证书的签名

CertificateFactory

package java.security.cert;
public class **CertificateFactory** 证书工厂
public static final CertificateFactory getInstance(String type) throws CertificateException 获取一个证书工厂实例 // CertificateFactory.getInstance("X.509");
public final Collection<? extends Certificate> generateCertificates(InputStream inStream) throws CertificateException 生成一个证书

RSAPublicKeySpec

package java.security.spec;
public class **RSAPublicKeySpec** implements KeySpec RSA 公钥
public RSAPublicKeySpec(BigInteger modulus, BigInteger publicExponent)
public BigInteger getModulus() 获取模数
public BigInteger getPublicExponent() 获取公共指数

X500Principal

package javax.security.auth.x500;
public final class **X500Principal** implements Principal, java.io.Serializable X.500 证书颁发者/证书使用者 标识名（" CN=Digicert, OU=Digicert, O=Digicert, L=Linton, ST=Utah, C=US "）

public X500Principal(String name) 构建一个标识名
public abstract byte[] getTBSCertificate() throws CertificateEncodingException 获取 DER 编码的证书信息

java.sql

连接

DataSource

public interface DataSource 自定义数据库连接池

Connection getConnection() 获取内部自定义数据库连接成员

DriverManager

package java.sql;
public class **DriverManager** 驱动管理对象

public static synchronized void registerDriver(java.sql.Driver driver) 注册驱动
Class.forName("com.mysql.jdbc.Driver")
public static synchronized void registerDriver(java.sql.Driver driver, DriverAction da)
private static Connection getConnection(String url, java.util.Properties info, Class<?> caller) 获取数据库连接
public static Connection getConnection(String url, String user, String password)

Connection

package java.sql;
public interface **Connection** extends Wrapper, AutoCloseable 数据库连接

Statement createStatement() 获取执行者对象
PreparedStatement prepareStatement(String sql) 获取预编译执行者对象（对 sql 进行预编译，明确 sql 的格式后，就不会改变了，

避免 sql 注入攻击）
void setAutoCommit(boolean autoCommit)　　参数为 false 开启事务
void commit()　提交事务
void rollback()　回滚事务

数据类型

Timestamp

package java.sql;
public class **Timestamp** extends java.util.Date　　时间戳类型（timestamp）
public Timestamp(int year, int month, int date, int hour, int minute, int second, int nano)　时间数字
public Timestamp(long time)　时间戳类型

public static Timestamp from(Instant.now())　获取时间 2021-03-31 04:08:28
public static Timestamp valueOf(String s)　转换为时间，格式必须为 yyyy-[m]m-[d]d hh:mm:ss[.f...]　// "2018-05-18 09:32:32"
public String toString()　　转换为字符串，格式为　yyyy-mm-dd hh:mm:ss.fffffffff　// 2022-03-02 11:22:33.0

Date

package java.sql;
public class **Date** extends java.util.Date　java.sql.Date 只是一个 java.util.Date，其时间设置为 yyyy-mm-dd
public static Date valueOf(String s)　　转换为日期，格式必须为 yyyy-[m]m-[d]d　// 2222-12-22　2222-1-2
public String toString ()　　转换为字符串，格式为 yyyy-mm-dd

Time

package java.sql;　（datetime）
public static Time valueOf(String s)　时间，格式必须为　hh:mm:ss　//　3:3:3
public String toString ()　　转换为字符串，格式为 hh:mm:ss

数据操作

Statement

package java.sql;
public interface **Statement** extends Wrapper, AutoCloseable

ResultSet executeQuery(String sql)　　执行查询
void addBatch(String sql) throws SQLException; 将 sql 添加到批处理中，等待统一执行（批处理不支持 select）
int[] executeBatch() throws SQLException;　　统一执行 sql（缓存了多个 Statement，等待 executeBatch 逐一执行）

PreparedStatement

package java.sql;
public interface **PreparedStatement** extends Statement　预编译处理

ResultSet executeQuery() throws SQLException;

ResultSet

package java.sql;
public interface **ResultSet** extends Wrapper, AutoCloseable

boolean next()
String getString(int columnIndex)　　根据列名索引获取数据

DatabaseMetaData

package java.sql;
public interface **DatabaseMetaData** extends Wrapper　　数据库的源信息

String getDatabaseProductName()　　获取数据库产品名称
String getDatabaseProductVersion()　　获取数据库产品版本号

ParameterMetaData

package java.sql;
public interface **ParameterMetaData** extends Wrapper　　参数的源信息

int getParameterCount() 获取 sql 中参数的个数

ResultSetMetaData

package java.sql;
public interface **ResultSetMetaData** extends Wrapper

int getColumnCount()　　获取列总数
String getColumnName(int column)　　获取列名

实例

jdbc 连接参数：　jdbc:mysql://192.168.22.129:8083/test_spring?verifyServerCertificate=false

user=xxx　　用户名
password=xxx　　密码
autoReconnect=true　　联机失败,是否重新联机
maxReconnect=3　　尝试重新联机次数
initialTimeout=3　　尝试重新联机间隔
maxRows=200　　传回最大行数
useUnicode=true　　是否使用 Unicode 字体编码
characterEncoding=UTF-8　　何种编码，GB2312/UTF-8
relaxAutocommit=true　　是否自动提交
capitalizeTypeNames=true　　数据定义的名称以大写表示
userSSL=false　　是否使用 ssl 连接

手动连接数据库：

```
public static void connectDb(){
    //声明驱动类
    Class jdbcDriverClass;
    //声明数据库连接
    Connection conn = null;
    try {
        //加载驱动类
        jdbcDriverClass = Class.forName("com.mysql.cj.jdbc.Driver");
        //创建驱动
        Driver driver = (Driver) jdbcDriverClass.newInstance();
        //注册驱动
        DriverManager.registerDriver(driver);
        //创建连接
```

```
            conn = (Connection)
DriverManager.getConnection("jdbc:mysql://192.168.22.129:8083/test_spring","labuladuo","balabala@@");
            Statement statement=conn.createStatement();
            ResultSet resultSet=statement.executeQuery("select * from student;");
            resultSet.next();
            System.out.println(resultSet.getString(2));
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

(exception)

SQLRecoverableException

package java.sql;

public class **SQLRecoverableException** extends java.sql.SQLException　　重试事务分支时，在以前失败的操作可能成功的情况下，抛出的 SQLException 的子类。恢复操作至少必须包括关闭当前连接和获取新连接

SQLException

package java.sql;

public class **SQLException** extends java.lang.Exception

implements Iterable<Throwable>

java.sql.SQLException: Fail to convert to internal representation　　When the return result order does not match, an error will be reported

java.time

Instant

package java.time;

public final class **Instant**　implements Temporal, TemporalAdjuster, Comparable<Instant>, Serializable

时间类更新：

java.util.Date　　==>　java.time.Instant

java.sql.Timestamp ==>　java.time.Instant

java.sql.Date　　==>　java.time.LocalDate

java.sql.Time　　==>　java.time.LocalTime

public static final Instant EPOCH = new Instant(0, 0);　　时间原点 1970-01-01T00:00:00Z

public static final Instant MAX = Instant.ofEpochSecond(MAX_SECOND, 999_999_999);　　时间最大值　+1000000000-12-31T23:59:59.999999999Z

public static final Instant MIN = Instant.ofEpochSecond(MIN_SECOND, 0);　　时间最小值　-1000000000-01-01T00:00:00Z

private final long seconds;　从时间原点 纪元 1970-01-01T00:00:00Z 开始的 秒数，时间戳

private final int nanos;　　从秒字段开始沿时间线的纳秒数。它总是正的，并且永远不会超过 999 999 999。

public static Instant now()　返回当前的 Instant 对象，这个构造方法不够精准，无法到纳秒级。

public static Instant ofEpochSecond(　　纳秒级时间对象

long epochSecond,　　从 1970-01-01T00:00:00Z 开始的秒数

long nanoAdjustment　　对秒数的纳秒调整，无论是正的还是负的

)

public static Instant parse(final CharSequence text)　读取字符串为时间对象，格式必须为：2007-12-03T10:15:30.00Z

public long getEpochSecond() 获取时间戳 // 155 557 4669
public int getNano() 获取纳秒数 // 633000000
public long toEpochMilli() 从时间原点 纪元 1970-01-01T00:00:00Z 开始的 毫秒数，时间戳 // 155 557 4669 633
public int compareTo(Instant otherInstant) 比较两个时间点
public boolean isAfter(Instant otherInstant) 是否在时间点之后
public boolean isBefore(Instant otherInstant) 是否在时间点之前
public Instant plusSeconds(long secondsToAdd) 加上秒数
public Instant plus(long amountToAdd, TemporalUnit unit) 加上指定类型的时间

LocalDate

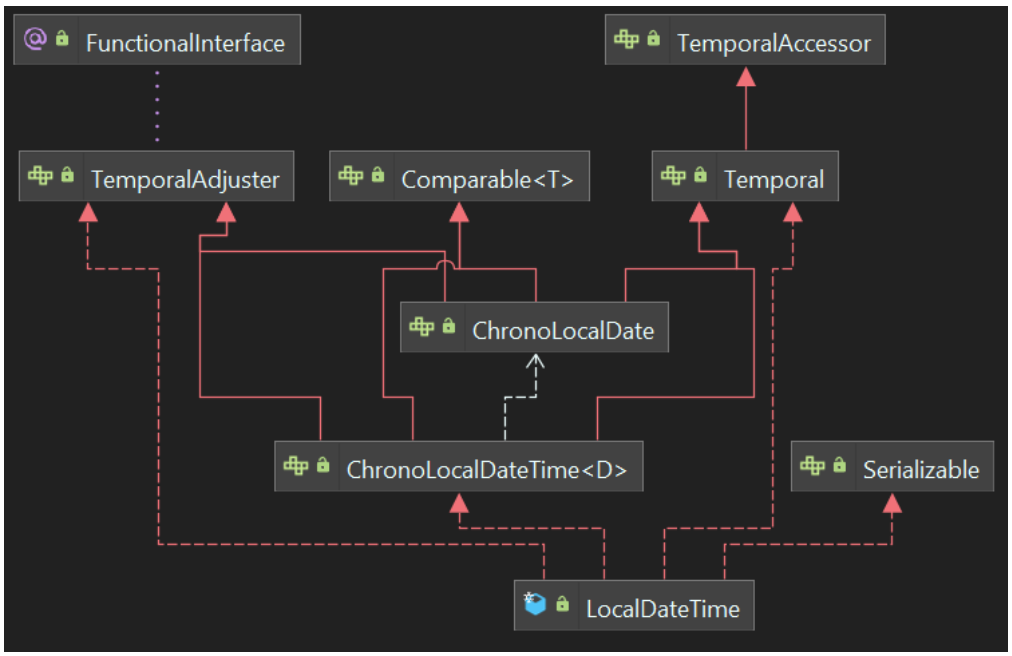
package java.time;
public final class **LocalDate** implements Temporal, TemporalAdjuster, ChronoLocalDate, Serializable
    An immutable date-time object representing a date, expressed as year, month, and day (e.g., 2010-10-10).
    It does not include any time information (such as hours, minutes, seconds).
    When performing addition or subtraction operations on a LocalDate object, the result is always a new immutable date object.

public String format(DateTimeFormatter formatter)
public static LocalDate now()
public static LocalDate of(int year, Month month, int dayOfMonth)
public static LocalDate parse(CharSequence text)
    Parses a string in the default ISO-8601 format to obtain a LocalDate object.
    // LocalDate.parse("2023-03-01");
public static LocalDate parse(CharSequence text, DateTimeFormatter formatter)
    Parses a string using the specified format to obtain a LocalDate object.
public static LocalDate now(ZoneId zone)
    Creates a LocalDate object for the current date in the specified time zone.
    // LocalDate.now(ZoneId.of("Asia/Shanghai"));
public int getYear()
public int getMonthValue()
public Month getMonth()
public DayOfWeek getDayOfWeek()
public int getDayOfMonth()
    Returns the day of the month, ranging from 1 to 31
public int getDayOfYear()
    Returns the day of the year, ranging from 1 to 365 (or 366 in a leap year)
public LocalDate withYear(int year)
    Returns a new LocalDate with the specified year
public LocalDate withMonth(int month)
    Returns a new LocalDate with the specified month
public LocalDate withDayOfMonth(int dayOfMonth)
    Returns a new LocalDate with the specified day of the month
public LocalDate withDayOfYear(int dayOfYear)
    Returns a new LocalDate with the specified day of the year
public LocalDate plusYears(long yearsToAdd)
public LocalDate minusYears(long yearsToSubtract)
public LocalDate plusMonths(long monthsToAdd)

public LocalDate minusMonths(long monthsToSubtract)
public LocalDate plusWeeks(long weeksToAdd)
public LocalDate minusWeeks(long weeksToSubtract)
public LocalDate plusDays(long daysToAdd)
public LocalDate minusDays(long daysToSubtract)
public boolean isEqual(ChronoLocalDate other
public boolean isBefore(ChronoLocalDate other)
public boolean isAfter(ChronoLocalDate other)
public int compareTo(ChronoLocalDate other)
public String toString()
Returns the date in the default ISO-8601 format (yyyy-MM-dd).

LocalDateTime

package java.time;
public final class **LocalDateTime** 本地日期时间
implements Temporal, TemporalAdjuster, ChronoLocalDateTime<LocalDate>, Serializable
一个不可变的时间日期时间对象，在对 LocalDateTime 对象进行 加减修改操 作后，返回的都是一个不可变的新日期时间对象



public static LocalDateTime now() 从系统中获取当前时间返回实例
public static LocalDateTime of(int year, Month month, int dayOfMonth, int hour, int minute) 构造一个本地日期时间
public static LocalDateTime of(LocalDate date, LocalTime time) 合并两个时间
public static LocalDateTime now(Clock clock) 从指定时钟获取当前日期时间
public int getHour() 一天中的时间，从 0 到 23
public int getMinute() 小时的分钟，从 0 到 59
public int getSecond() 分钟的秒，从 0 到 59
public int getNano() 秒的纳秒，从 0 到 999999999

LocalTime

package java.time;
public final class **LocalTime**
implements Temporal, TemporalAdjuster, Comparable<LocalTime>, Serializable 本地时间

Duration

package java.time;
public final class **Duration**　　持续时间
　　implements TemporalAmount, Comparable<Duration>, Serializable
public static Duration ofMillis(long millis)　　获取表示多个标准小时数的持续时间
public static Duration ofMinutes(long minutes)　　此方法返回以 1 分钟格式表示时间的 Duration

ZoneId

package java.time;
public abstract class **ZoneId** implements Serializable　时区 id
public static ZoneId of(String zoneId)　　获取指定时区　// ZoneId.of("Asia/Shanghai")
　　GMT　　前世界标准时，
　　UTC　　现世界标准时。UTC 比 GMT 更精准，以原子时计时，适应现代社会的精确计时。
　　GMT+8　东八区的时间

public static Set<String> getAvailableZoneIds()　获取地球所有时区 id

ZonedDateTime

package java.time;
public final class **ZonedDateTime** implements Temporal, ChronoZonedDateTime<LocalDate>, Serializable

public static ZonedDateTime now()
　　Obtains the current date-time from the system clock in the default time-zone.

Month

package java.time;
public enum **Month** implements TemporalAccessor, TemporalAdjuster　月份抽象
public Month firstMonthOfQuarter()　获取与本季度的第一个月对应的月份
public int minLength()　本月最少天数
public int getValue()　　获取值，一年中的月份，从 1 月 1 日到 12 月 12 日

Clock

package java.time;
public abstract class **Clock** implements InstantSource　时钟
public static Clock systemUTC()　获取 UTC 时区时钟

DayOfWeek

package java.time;

public enum **DayOfWeek** implements TemporalAccessor, TemporalAdjuster　　星期几

public int getValue()　从 1（星期一）到 7（星期日）

temporal

TemporalUnit

package java.time.temporal;

public interface **TemporalUnit**　临时单位

ChronoUnit

package java.time.temporal;

public enum **ChronoUnit** implements TemporalUnit　计时单位

NANOS("Nanos", Duration.ofNanos(1))　　纳秒

MICROS("Micros", Duration.ofNanos(1000))　　微秒

MILLIS("Millis", Duration.ofNanos(1000_000))　毫秒

SECONDS("Seconds", Duration.ofSeconds(1))　秒

MINUTES("Minutes", Duration.ofSeconds(60))　分钟

HOURS("Hours", Duration.ofSeconds(3600))　　小时

HALF_DAYS("HalfDays", Duration.ofSeconds(43200))　　半天

DAYS("Days", Duration.ofSeconds(86400))　　天数

WEEKS("Weeks", Duration.ofSeconds(7 * 86400L))　　星期

MONTHS("Months", Duration.ofSeconds(31556952L / 12))　一个月

YEARS("Years", Duration.ofSeconds(31556952L))　　年

DECADES("Decades", Duration.ofSeconds(31556952L * 10L))　10 年

CENTURIES("Centuries", Duration.ofSeconds(31556952L * 100L))　100 年

MILLENNIA("Millennia", Duration.ofSeconds(31556952L * 1000L))　1000 年

ERAS("Eras", Duration.ofSeconds(31556952L * 1000_000_000L))　　一个纪元

FOREVER("Forever", Duration.ofSeconds(Long.MAX_VALUE, 999_999_999))　永远

TemporalAdjuster

package java.time.temporal;

public interface **TemporalAdjuster**　时间调整器

Temporal adjustInto(Temporal temporal);　将一个时间调整一下

public long between(Temporal temporal1Inclusive, Temporal temporal2Exclusive)　获取两个时间之前的 指定单位数量

获取两个日期之间的每一天

```
LocalDate startDate = LocalDate.parse("2021-09-01");
LocalDate endDate = LocalDate.parse("2021-10-20");

List<LocalDate> gapDays = new ArrayList<>();
Stream.iterate(endDate, d -> d.minusDays(1)).limit(ChronoUnit.DAYS.between(startDate, endDate) + 1).forEach(gapDays::add);
System.out.println(gapDays);
```

format

DateTimeFormatter

package java.time.format;

public final class **DateTimeFormatter**

Formatter for printing and parsing date-time objects.

public static final DateTimeFormatter ISO_LOCAL_DATE_TIME;　　// 2023-03-03T10:36:08.72

public static DateTimeFormatter ofPattern(String pattern)

Create a formater using the specified pattern.

```
Timestamp timestamp = new Timestamp(System.currentTimeMillis());
System.out.println(timestamp.toLocalDateTime().format(DateTimeFormatter.ofPattern("yyyy-MM-dd
HH:mm:ss.SSS")));
```

public String format(TemporalAccessor temporal)

Formats a date-time object using this formater.

java.text

DateFormat

package java.text;

public abstract class **DateFormat** extends Format

public Date parse(String source) throws ParseException 解析字符串返回日期对象 // parse("2021-08-06 00:00:01")

public final String format(Date date) 格式化日期，返回字符串

SimpleDateFormat

package java.text;

public class **SimpleDateFormat** extends DateFormat

Thread-safe

SimpleDateFormat is not thread-safe because it maintains internal state, such as the date format pattern and locale, that can be modified during parsing or formatting operations.

When multiple threads use the same instance of SimpleDateFormat, this shared state can lead to inconsistent results, data corruption, or other unexpected behaviors.

Year, Month, Day:

yyyy: Year (e.g., 2024)

MM: Month (01–12)

// "yyyy-MM-dd" - "2024-3-25" ✗

M: Month (1–12)

// "yyyy-M-dd" - "2024-02-25" ✓

dd: Day of the month (01–31)

// "yyyy-MM-dd" - "2024-03-2" ✗

d: Day of the month without leading zeros (1–31)

// "yyyy-M-d" - "2024-02-05" ✓

"MMMM dd, yyyy", Locale.ENGLISH (e.g., September 29, 2013)

Hour, Minute, Second:

hh: Hour in 12-hour format (01–12)

h: Hour in 12-hour format without leading zeros (1–12)

HH: Hour in 24-hour format (00–23)

mm: Minutes (00–59)

ss: Seconds (00–59)

SSS: Milliseconds (000–999)

Week and Day:
EEE: Day of the week (short format, e.g., Tue)
EEEE: Day of the week (full format, e.g., Tuesday)
D: Day of the year (1–365/366)
w: Week number of the year (1–52/53)
W: Week number of the month (1–5)

AM/PM and Miscellaneous:
a: AM/PM marker (e.g., AM, PM)
k: 24-hour format (1–24)
K: 12-hour format (0–11)

public SimpleDateFormat(String pattern)

private transient char[] compiledPattern;

public Date parse(String text, ParsePosition pos)
public StringBuffer format(Date date, StringBuffer toAppendTo, FieldPosition pos)

NumberFormat

package java.text;
public abstract class **NumberFormat** extends Format 数字格式

public final static NumberFormat getPercentInstance() 获取百分比格式化对象
public final static NumberFormat getNumberInstance() 获取数字格式化对象
public void setMaximumIntegerDigits(int newValue) 设置数的整数部分所允许的最大位数
public void setMinimumIntegerDigits(int newValue) 设置数的 整数部分 所允许的最小位数

public void setMaximumFractionDigits(int newValue) 设置最多保留小数位数，不足不补 0 //
nf.setMaxmumFractionDigits(2) nf.format(3.66666)
public void setMinimumFractionDigits(int newValue) 设置最少小数点位数，不足的位数以 0 补位，超出的话按实际位数输出。

public void setGroupingUsed(boolean newValue) 设置是否分组

public final String format(double number) 格式化小数

DecimalFormat

package java.text;
public class **DecimalFormat** extends NumberFormat

public DecimalFormat(String pattern)
new DecimalFormat("#.##").format(345.6666666) 345.67
new DecimalFormat("#.##").format(0.66666) 0.67
skips leading/trailing zeros.
new DecimalFormat("##,##.##").format(345.6666666) 3,45.67
new DecimalFormat("##,##.##").format(0.6666666) 0.67

‘,’ groups digits every two places.

new DecimalFormat("0.00").format(345.66666) 345.67
new DecimalFormat("0.00").format(0.66666) 0.67
new DecimalFormat("00,00.00").format(345.6666666) 03,45.67
new DecimalFormat("00,00.00").format(0.6666666) 00,00.67
‘0’ doesn’t skip leading/trailing zeros.

new DecimalFormat("#.00").format(345.66666) 345.67
new DecimalFormat("#.00").format(0.66666) .67

java.util
(Data Type)
(Queue)

Queue

package java.util;
public interface **Queue**<E> extends Collection<E>
Convert a Queue<Character> to a String
Queue<Character> strQueue=new LinkedList<>();
String resultString = strQueue.stream().map(String::valueOf).collect(Collectors.joining());

All Queue Data Structures in java:
java.util.LinkedList
java.util.PriorityQueue
java.util.ArrayDeque

java.util.concurrent.ArrayBlockingQueue
java.util.concurrent.LinkedBlockingQueue
java.util.concurrent.LinkedBlockingDeque
java.util.concurrent.LinkedTransferQueue
java.util.concurrent.PriorityBlockingQueue
java.util.concurrent.ConcurrentLinkedQueue

java.util.concurrent.DelayQueue
java.util.concurrent.SynchronousQueue

boolean offer(E e);
Inserts the specified element into the tail of this queue if it is possible to do so immediately without violating capacity restrictions.
When using a capacity-restricted queue, this method is generally preferable to add, which can fail to insert an element only by throwing an exception.
PriorityQueue:
Efficient O(log n) insert/remove.
E poll()
Retrieves and removes the head of this queue, or returns null if this queue is empty.
E peek();
Retrieves, but does not remove, the head of this queue, or returns null if this queue is empty.

Stack.peek() in java.util.Stack will throw an EmptyStackException if the stack is empty.

E remove();
Retrieves and removes the head of this queue.
This method differs from poll() only in that it throws an exception if this queue is empty.
E element();
Retrieves, but does not remove, the head of this queue. This method differs from peek only in that it throws an exception if this queue is empty.

boolean add(E e);
Inserts the specified element into the tail of this queue if it is possible to do so immediately without violating capacity restrictions,
returning true upon success and throwing an IllegalStateException if no space is currently available.

Deque

package java.util;
public interface **Deque**<E> extends Queue<E> Double ended queue, FIFO (First-In-First-Out) behavior results.

boolean offerFirst(E e);
boolean offerLast(E e);
E pollFirst();
E pollLast();
E peekFirst();
E peekLast();
The method can return null.

void addFirst(E e);
void addLast(E e);

Iterator<E> iterator();
The elements will be returned in order from first (head) to last (tail).
Iterator<E> descendingIterator();
The elements will be returned in order from last (tail) to first (head).

ArrayDeque

package java.util;
public class **ArrayDeque**<E> extends AbstractCollection<E> implements Deque<E>, Cloneable, Serializable

1) Null Elements
   ArrayDeque does not permit null elements. Any attempt to insert a null element will result in a NullPointerException.
2) Constant Time
   The operations addFirst, addLast, removeFirst, removeLast, getFirst, and getLast typically run in constant time, O(1).
   This means that these operations are highly efficient and can be expected to execute quickly regardless of the number of elements in the deque.
   `java.util.ArrayDeque` offers better performance than `java.util.Stack` as `Stack` is synchronized and incurs additional locking overhead.
3) Capacity

The ArrayDeque class automatically resizes itself as elements are added.

It uses a resizable array to store the elements, and when the array becomes full, it allocates a larger array and moves the elements to the new array. This ensures that the deque can grow to accommodate any number of elements.

4) Order of Elements

ArrayDeque maintains the order of elements based on their insertion. Elements added to the front or the back of the deque are kept in that order unless they are removed.

This makes it suitable for applications that require elements to be processed in a specific order, such as a queue or stack.

PriorityQueue

public class **PriorityQueue**<E> extends AbstractQueue<E>
implements java.io.Serializable

1) Null Elements

PriorityQueue does not permit null elements. Any attempt to insert a null element will result in a NullPointerException.

2) Constant Time

The operations peek and element (to access the head of the queue) typically run in constant time, O(1).

However, the add, offer, remove, and poll operations typically run in logarithmic time, O(log n), , where n is the number of elements in the priority queue.

3) Capacity

A PriorityQueue grows automatically as elements are added.

It uses a resizable array to store the elements. When the array becomes full, it allocates a larger array and moves the elements to the new array. This ensures that the queue can grow to accommodate any number of elements.

4) Order of Elements

The elements of the priority queue are ordered according to their natural ordering, or by a Comparator provided at queue construction time, depending on which constructor is used.

However, the main drawback is that it doesn't allow random access to elements, which makes querying less efficient.

transient Object[] queue; // non-private to simplify nested class access

PriorityQueue uses an array to store elements. The min-heap structure is maintained within this array.

By default, a PriorityQueue in Java is implemented as a min-heap, which means:

The smallest element is always at the front of the queue (retrieved by peek() or poll()).

The internal structure of the heap does not guarantee that the largest element is at the end or in any specific position. calculations:

```
PriorityQueue<Integer> minHeap = new PriorityQueue<>(); // Min-heap
PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a); // Max-heap
```

The largest element will always be at the front (head) of the Max-heap.

In a min-heap stored in an array, the parent-child relationships can be easily determined using simple mathematical

For a node at index `i`

Parent: Located at index `(i-1)/2`

Left Child: Located at index `2i+1`

Right Child: Located at index `2i+2`

Example

Let's consider a binary heap with 7 elements: [10, 20, 30, 40, 50, 60, 70].

Index 0:

Element: 10

Left Child: 2*0 + 1 = 1 (Element: 20)

Right Child: 2*0 + 2 = 2 (Element: 30)

Index 1:
Element: 20
Left Child: 2*1 + 1 = 3 (Element: 40)
Right Child: 2*1 + 2 = 4 (Element: 50)
Index 2:
Element: 30
Left Child: 2*2 + 1 = 5 (Element: 60)
Right Child: 2*2 + 2 = 6 (Element: 70)
Visualization

```
        10 (index 0)
       /   \
   20 (1)  30 (2)
   / \     / \
40(3)50(4)60(5)70(6)
```

private static final int DEFAULT_INITIAL_CAPACITY = 11;
public PriorityQueue(Comparator<? super E> comparator)
// `PriorityQueue<Integer> heap = new PriorityQueue<>((o1, o2) -> map[o2] - map[o1]);`
By default, PriorityQueue is a min-heap, meaning the smallest element is always at the head.
poll() → Retrieves and removes the head (minimum element).

public PriorityQueue()
public PriorityQueue(int initialCapacity)
Creates a PriorityQueue with the specified initial capacity that orders its elements according to their natural ordering.
Internally, the PriorityQueue uses an array to store its elements.
When the number of elements exceeds the current capacity of the array, it will resize the array automatically to accommodate more elements.
If you set an initial capacity for the PriorityQueue, the time complexity of the offer method will consistently be $O(\log n)$, where n is the number of elements in the priority queue.
public PriorityQueue(int initialCapacity, Comparator<? super E> comparator)

Stack

public class **Stack**<E> extends Vector<E>

1) Null Elements
   Stack permits null elements. This means you can push null values onto the stack, and they will be treated as valid elements.
2) Constant Time
   The operations push, pop, peek, empty, and search typically run in constant time, O(1). This makes these operations highly efficient and quick to execute.
   `java.util.ArrayDeque` offers better performance than `java.util.Stack` as `Stack` is synchronized and incurs additional locking overhead.
3) Capacity
   As elements are added to a Stack, its capacity grows automatically. The Stack class, which extends Vector, can dynamically resize itself to accommodate new elements.
4) Order of Elements
   Stack maintains the order of elements based on a Last-In-First-Out (LIFO) principle. The most recently added element is the first to be removed.

public E push(E item)
public synchronized E pop()
public synchronized E peek()
public boolean empty()
    Verify if the queue is empty
Override
public synchronized int size()

(List)

List

package java.util;
public interface **List**<E> extends Collection<E>

Convert primitive array to List

int[] -> Modifiable List

```
List<Integer> integerList = Arrays.stream(primitiveArray).boxed().collect(Collectors.toList());
```

int[] -> Immutable List

```
List<Integer> integerList = Arrays.stream(primitiveArray).boxed().toList();
```

int[] -> List

```
int[] primitiveArray = {1, 2, 3, 4, 5};

List<Integer> arrayList = new ArrayList<>();
for (int value : primitiveArray) {
    arrayList.add(value);
}
```

List -> int[]

```
int[] primitiveArray = list.stream().mapToInt(Integer::intValue).toArray();
```

List -> int[]

```
int[] primitiveArray = new int[list.size()];
for (int i = 0; i < list.size(); i++) {
    primitiveArray[i] = list.get(i); // Auto-unboxing happens here
}
```

int[] -> Integer[]

```
int[] primitiveArray = {1, 2, 3, 4, 5};

Integer[] integerArray = new Integer[primitiveArray.length];
for (int i = 0; i < primitiveArray.length; i++) {
    integerArray[i] = primitiveArray[i]; // Autoboxing to Integer
}

// Use Arrays.asList to create a List<Integer> from Integer[]
List<Integer> integerList = Arrays.asList(integerArray);
```

Integer[] -> int[]

```
arrays.stream(objectArray).mapToInt(Integer::intValue).toArray()
```

"aaa"=> Character[]

```
Character[] array = randomString.chars().mapToObj(cr->(char)cr).toArray(Character[]::new);
```

Apache Commons Lang

```
import org.apache.commons.lang3.ArrayUtils;

import java.util.Arrays;
import java.util.List;

public class PrimitiveToListWithCommonsLang {
    public static void main(String[] args) {
        int[] primitiveArray = {1, 2, 3, 4, 5};

        // Convert int[] to Integer[] using Apache Commons Lang ArrayUtils
```

```
                Integer[] integerArray = ArrayUtils.toObject(primitiveArray);

                // Convert Integer[] to List<Integer> using Arrays.asList
                List<Integer> integerList = Arrays.asList(integerArray);

                // Output the result
                System.out.println(integerList);
            }
        }
```

```
boolean add(E e);
void add(int index, E element);
    Add a new element before the element at `index` and the remaining elements
boolean addAll(Collection<? extends E> c);                  末尾追加 List
boolean addAll(int index, Collection<? extends E> c);     索引 index 添加 list，排开包括索引 index 和它后方的元素
boolean remove(Object o);                                  删除索引 o 的数据 或  删除值为 o 的数据（后方元素前移）       // remove(2)
remove("xx")
boolean removeAll(Collection<?> c);          删除值存在于 c 内部的数据（多个相同的值存在于原始 list 内会被全部删除）
E get(int index);
E set(int index, E element);

int size();
boolean isEmpty();
List<E> subList(int fromIndex, int toIndex)
    Get a sublist [ 2, 9 )
    Time Complexity: O(1) (amortized)
    fromIndex and toIndex must be within the given list index range. otherwise an IndexoutOfBoundsException will be thrown.
Object[] toArray();
default void sort(Comparator<? super E> c)
        listP.sort((x1,x2)->x1.getName().compareTo(x2.name))
        listP.sort((x1,x2)-> {
                if(x1.getWeight()>x2.getWeight()){
                    return 1;
                } else if(x1.getWeight()<x2.getWeight()){
                    return -1;
                }else{
                    return 0;
                }
        });
Arrays.stream(nameList)
default Spliterator<E> spliterator()    返回分流器
boolean tryAdvance(Consumer<? super T> action)     组合了迭代器的 hasNext()和 next() 方法
default void forEachRemaining(Consumer<? super T> action)     遍历剩下的元素
Spliterator<T> trySplit();
boolean contains(Object o);
Object[] toArray();

boolean remove(Object o);
E remove(int index);
```

Removes the element at the specified position in this list (optional operation).

static <E> List<E> copyOf(Collection<? extends E> coll)
Returns an unmodifiable list containing the elements of the given Collection.

static <E> List<E> of()
List.of
Returns an immutable list.
Any attempt to modify the list (e.g., add, remove, set) will throw an UnsupportedOperationException.
Arrays.asList
Returns a fixed-size list backed by the original array.
The list is modifiable in terms of updating elements (e.g., set), but you cannot add or remove elements (throws UnsupportedOperationException).

static <E> List<E> of(E e1)
static <E> List<E> of(E e1, E e2)
static <E> List<E> of(E e1, E e2, E e3)
static <E> List<E> of(E e1, E e2, E e3, E e4, E e5, E e6, E e7, E e8, E e9, E e10)
static <E> List<E> of(E... elements)

AbstractList

package java.util;
public abstract class **AbstractList**<E> extends AbstractCollection<E> implements List<E>
A normal Iterator in Java does not support adding elements to a collection.
To add elements at a specific position while iterating through a collection, you need to use a ListIterator, which is a more advanced form of an iterator.

java.util.ArrayList
java.util.LinkedList
protected transient int modCount = 0;
public ListIterator<E> listIterator()
void add(E e);
The element is inserted before the cursor position.

Collection

Reference:
https://www.baeldung.com/java-fail-safe-vs-fail-fast-iterator

package java.util;
public interface **Collection**<E> extends Iterable<E>
Fail-Fast Iterators
Abort operation
Fail-Fast systems abort operation as-fast-as-possible exposing failures immediately and stopping the whole operation.
When traversing a collection object with an iterator, if the content of the collection object is modified (added, deleted, modified) during the traversal process, a ConcurrentModifications Exception will be thrown

The Fail-Fast behavior isn't guaranteed to happen in all scenarios as it's impossible to predict behavior in case of concurrent modifications.

These iterators throw ConcurrentModificationException on a best effort basis.

Example:

```
ArrayList<Integer> numbers = // ...

Iterator<Integer> iterator = numbers.iterator();
while (iterator.hasNext()) {
    Integer number = iterator.next();
    numbers.add(50);
}
// In the code snippet above, the ConcurrentModificationException gets thrown at the beginning of a next iteration cycle after the modification was performed.
```

Classes

Default iterators for Collections from java.util package are Fail-Fast.

java.util.ArrayList

java.util.LinkedList

java.util.HashSet

java.util.LinkedHashSet

java.util.TreeSet

java.util.HashMap

java.util.LinkedHashMap

java.util.TreeMap

modCount

Collections maintain an internal counter called modCount. Each time an item is added or removed from the Collection, this counter gets incremented.

When iterating, on each next() call, the current value of modCount gets compared with the initial value. If there's a mismatch,

it throws ConcurrentModificationException which aborts the entire operation.

The remove method

If during iteration over a Collection, an item is removed using Iterator's remove() method, that's entirely safe and doesn't throw an exception.

```
ArrayList<Integer> numbers = // ...

Iterator<Integer> iterator = numbers.iterator();
while (iterator.hasNext()) {
    if (iterator.next() == 30) {
        iterator.remove(); // ok!
    }
}

iterator = numbers.iterator();
while (iterator.hasNext()) {
    if (iterator.next() == 40) {
        numbers.remove(2); // exception
    }
}
```

Fail-Safe Iterators

abort operation

Fail-Safe systems don't abort an operation in the case of a failure. Such systems try to avoid raising failures as much as possible.

The clone of the actual Collection

Those iterators create a clone of the actual Collection and iterate over it.

If any modification happens after the iterator is created, the copy still remains untouched. Hence, these Iterators continue looping over the Collection even if it's modified.

disadvantage

The Fail-Safe Iterators have a few disadvantages, though. One disadvantage is that the Iterator isn't guaranteed to return updated data from the Collection,

as it's working on the clone instead of the actual Collection.

Another disadvantage is the overhead of creating a copy of the Collection, both regarding time and memory.

classes

Iterators on Collections from java.util.concurrent package such as ConcurrentHashMap, CopyOnWriteArrayList, etc. are Fail-Safe in nature.

```
default Stream<E> stream()
    // List<String> names = userList.stream().map( usera -> 22 ) .collect(Collectors.toList())

default Stream<E> parallelStream()

Iterator<E> iterator();
Object[] toArray();
<T> T[] toArray(T[] a);
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c);
    Adds all of the elements in the specified collection to this collection.
boolean removeAll(Collection<?> c);
default boolean removeIf(Predicate<? super E> filter)
boolean retainAll(Collection<?> c);
void clear();
int size();
    The time complexity of the size() method in the Collection<E> interface (from Java's standard library) depends on the specific implementation of the collection.
boolean isEmpty();
default Stream<E> stream()
    Collection<Integer> -> int[]
        int[] result = queue.stream().mapToInt(Integer::intValue).toArray();
int hashCode();
boolean equals(Object o);
```

ArrayList

```
package java.util;
public class ArrayList<E>
        extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```

1) Null Elements

ArrayList permits null elements. This means you can add null values to an ArrayList, and they will be treated as valid elements.

2) Constant Time

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time, O(1).

The add operation runs in amortized constant time, meaning that occasional resizing can cause the add operation to take linear time, O(n), but the average time remains constant.

Creating a new array with a larger capacity.

Copying all the elements from the old array to the new array.

Adding the new element to the new array.

The remove and contains operations run in linear time, O(n), as remove may require shifting elements and contains requires checking each element.

3) Capacity

As elements are added to an ArrayList, its capacity grows automatically. It uses a resizable array to store the elements,

and when the array becomes full, it allocates a larger array (usually 1.5 times the size of the original) and moves the elements to the new array.

This ensures that the ArrayList can grow to accommodate any number of elements.

```
private Object[] grow() {
    return grow(size + 1);
}

private Object[] grow(int minCapacity) {
    int oldCapacity = elementData.length;
    if (oldCapacity > 0 || elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        int newCapacity = ArraysSupport.newLength(
                  oldCapacity,
                  minCapacity - oldCapacity,          /* minimum growth */
                  oldCapacity >> 1                    /* preferred growth */
         );
        return elementData = Arrays.copyOf(elementData, newCapacity);
    } else {
        return elementData = new Object[Math.max(DEFAULT_CAPACITY, minCapacity)];
    }
}
```

4) Order of Elements

ArrayList maintains the insertion order of elements. Elements are stored in the order they were added, and this order is preserved when iterating over the list or accessing elements by index.

private static final int DEFAULT_CAPACITY = 10;

private static final Object[] DEFAULTCAPACITY_EMPTY_ELEMENTDATA = {};

```
public ArrayList()
```

Constructs an empty list with an initial capacity of ten.

```
{
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}
```

```
public ArrayList(Collection<? extends E> c)
```

Params:

c – the collection whose elements are to be placed into this list

Throws:

NullPointerException – if the specified collection is null

This constructor can convert an immutable list into a modifiable one. The ArrayList itself is a new, independent,

modifiable collection

```
List<String> sourceList = List.of("Apple", "Banana", "Cherry");
ArrayList<String> arrayList = new ArrayList<>(sourceList);
```

Convert Set to List:

```
List<String> res = new ArrayList<>(hashset);
```

The constructor initializes the internal array and copies elements from c using toArray(), followed by System.arraycopy().
The toArray() method for most collection types takes O(n) time, where n is the number of elements in c.
The System.arraycopy() operation also takes O(n) time.
Thus, the overall time complexity is O(n), where n is the size of the input collection.

```
public ArrayList(int initialCapacity)
```

Constructs an empty list with the specified initial capacity.

Params:

initialCapacity – the initial capacity of the list

Throws:

IllegalArgumentException – if the specified initial capacity is negative

```
public void ensureCapacity(int minCapacity)
```

Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

Params:

minCapacity – the desired minimum capacity

```
private void add(E e, Object[] elementData, int s) {
  if (s == elementData.length)
    elementData = grow();
  elementData[s] = e;
  size = s + 1;
}

private Object[] grow() {
    return grow(size + 1);
}

private Object[] grow(int minCapacity) {
    int oldCapacity = elementData.length;
    if (oldCapacity > 0 || elementData != DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        int newCapacity = ArraysSupport.newLength(
                  oldCapacity,
                  minCapacity - oldCapacity,          /* minimum growth */
                  oldCapacity >> 1                    /* preferred growth */
         );
        return elementData = Arrays.copyOf(elementData, newCapacity);
    } else {
        return elementData = new Object[Math.max(DEFAULT_CAPACITY, minCapacity)];
    }
}

public Iterator<E> iterator() {
    return new Itr();
}
```

```
public boolean contains(Object o)

public int indexOf(Object o)
```

LinkedList

package java.util;
public class **LinkedList**<E>
extends AbstractSequentialList<E>
implements List<E>, Deque<E>, Cloneable, java.io.Serializable

1) Null Elements
   LinkedList permits all elements (including null).
2) Constant Time
   The size, isEmpty and iterator operations run in constant time, O(1).
   For listIterator, it is O(1) only if starting from the beginning. If starting from an index, it is O(n).

   The add and remove operations run in constant time when performed using iterators, but in linear time O(n)when performed by index.
   The get, set and contains operations run in linear time O(n) because accessing an element requires traversing the list.
3) Capacity
   LinkedList does not have a predefined capacity as it grows dynamically with the addition of each element. It is implemented as a doubly-linked list, which allows efficient insertion and removal of elements from both ends of the list.
4) Order of Elements
   LinkedList maintains the insertion order of elements. Elements are stored in the order they were added, and this order is preserved when iterating over the list or accessing elements sequentially.

```
public LinkedList()
public LinkedList(Collection<? extends E> c)
    new LinkedList<>(Arrays.asList(arr))
```

public boolean add(E e) add 是加在 list 尾部，push 施加在 list 头部. 等同于 addFirst
public void addFirst(E e)
public void addLast(E e)
public E getFirst()
public E getLast()
public E removeFirst()
public E removeLast()

Vector

package java.util;
public class **Vector**<E>
extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, java.io.Serializable

1) Null Elements
   Vector permits all elements, including null.
2) Constant Time

The size, isEmpty, get, set, iterator, and listIterator operations run in constant time, O(1).

The add operation in Vector runs in amortized constant time, similar to ArrayList.

The remove and contains operations run in linear time, O(n), as remove may require shifting elements and contains requires checking each element.

3) Capacity

As elements are added to an ArrayList, its capacity grows automatically. (see add method)

```
private Object[] grow() {
    return grow(elementCount + 1);
}

private Object[] grow(int minCapacity) {
    int oldCapacity = elementData.length;
    int newCapacity = ArraysSupport.newLength(
            oldCapacity,
            minCapacity - oldCapacity,                                  /* minimum growth */
            capacityIncrement > 0 ? capacityIncrement : oldCapacity     /* preferred growth *
    );
    return elementData = Arrays.copyOf(elementData, newCapacity);
}
```

4) Fail-fast

The iterators returned by this class's iterator and listIterator methods are fail-fast.

The Enumerations returned by the elements method are not fail-fast;

```
protected transient int modCount = 0;
protected int capacityIncrement;
protected Object[] elementData;


public Vector() {
    this(10);
}

public Vector(int initialCapacity, int capacityIncrement) {
    super();
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);
    this.elementData = new Object[initialCapacity];
    this.capacityIncrement = capacityIncrement;
}



public synchronized void insertElementAt(E obj, int index)
public synchronized void removeElementAt(int index)
public synchronized E elementAt(int index)
public synchronized boolean add(E e)
public synchronized void addElement(E obj) {
   modCount++;
   add(obj, elementData, elementCount);
}


public Enumeration<E> elements()
```

```
private void add(E e, Object[] elementData, int s) {
    if (s == elementData.length)
        elementData = grow();
    elementData[s] = e;
    elementCount = s + 1;
}

private Object[] grow() {
    return grow(elementCount + 1);
}

private Object[] grow(int minCapacity) {
    int oldCapacity = elementData.length;
    int newCapacity = ArraysSupport.newLength(
            oldCapacity,
            minCapacity - oldCapacity,                                  /* minimum growth */
            capacityIncrement > 0 ? capacityIncrement : oldCapacity     /* preferred growth */
    );
    return elementData = Arrays.copyOf(elementData, newCapacity);
}
```

(Set)

Set

package java.util;

public interface **Set**<E> extends Collection<E>

boolean add(E e);

void clear();

boolean remove(Object o);

Remove an element

boolean contains(Object o); 判断集合中是否包含某一个元素

boolean isEmpty(); 判断集合是否为空

nt size(); 返回集合的大小

Iterator<E> iterator(); 返回一个迭代器 从上到下扫描 // 遍历 while(iterator.hasNext()){E x=i.next()}

Object[] toArray(); 转换成数组

BitSet

package java.util;

public class **BitSet** implements Cloneable, java.io.Serializable

1) Null Elements

   BitSet does not allow null elements as it is a collection of bits, not objects.

2) Constant Time

   The size, isEmpty, get, set, and clear operations typically run in constant time, O(1).

3) Capacity

   BitSet dynamically grows as bits are set. There is no predefined capacity, and it automatically adjusts based on the highest bit set.

4) Insertion Order

   TreeSet maintains elements in a sorted order based on their natural ordering or by a comparator provided at set creation time. The iteration order of the elements is sorted and not based on the insertion order.

public BitSet(int nbits)

public void set(int bitIndex, boolean value)

public boolean get(int bitIndex)

HashSet

package java.util;

public class **HashSet**<E>　　基于 HashMap 实现，初始最大容量 16，初始容量=c.size()/.75f（因为无序，所以用 iterator 遍历）

extends AbstractSet<E>

implements Set<E>, Cloneable, java.io.Serializable

This class implements the Set interface, backed by a hash table (actually a HashMap instance).

It makes no guarantees as to the iteration order of the set; in particular, it does not guarantee that the order will remain constant over time.

1) Null Elements

HashSet permits null elements.

2) Constant Time

The size, isEmpty, contains, and remove operations typically run in constant time, O(1).

The add operation also typically runs in constant time, O(1), assuming a good hash function that disperses elements uniformly across buckets.

However, in the worst case (e.g., if many elements hash to the same bucket), the add operation can degrade to linear time.

3) Capacity

As elements are added to a `HashSet`, its capacity grows automatically.

4) Order of Elements

HashSet does not maintain any order of its elements. The iteration order of the elements is not guaranteed and can be different from the order in which they were added.

```
private transient HashMap<E,Object> map;

private static final Object PRESENT = new Object();     // Dummy value to associate with an Object in the backing Map

public HashSet(Collection<? extends E> c) {
   map = new HashMap<>(Math.max((int) (c.size()/.75f) + 1, 16));
   addAll(c);
}

public HashSet(int initialCapacity, float loadFactor) {
   map = new HashMap<>(initialCapacity, loadFactor);
}
```

Constructs a new, empty set; the backing HashMap instance has the specified initial capacity and the specified load factor.

Params:

initialCapacity – the initial capacity of the hash map

loadFactor – the load factor of the hash map

Throws:

IllegalArgumentException – if the initial capacity is less than zero, or if the load factor is nonpositive

```
public HashSet(int initialCapacity)
```

```
public boolean add(E e) {
    return map.put(e, PRESENT)==null;
}
```

Adds the specified element to this set if it is not already present. More formally, adds the specified element e to this set if this set contains no element e2 such that Objects.equals(e, e2).

If this set already contains the element, the call leaves the set unchanged and returns false.

Params:

e – element to be added to this set

Returns:

true if this set did not already contain the specified element

```
public Iterator<E> iterator()
```

LinkedHashSet

```
package java.util;
public class LinkedHashSet<E>
    extends HashSet<E>
    implements Set<E>, Cloneable, java.io.Serializable
```

1) Null Elements

   LinkedHashSet permits null elements.

2) Constant Time

   size, isEmpty, contains, and remove: These operations typically run in constant time, O(1).

   The add operation also typically runs in constant time, O(1), assuming a good hash function that disperses elements uniformly across buckets.

   However, in the worst case (e.g., if many elements hash to the same bucket), the add operation can degrade to linear time, O(n).

3) Capacity

   As elements are added to a LinkedHashSet, its capacity grows automatically.

4) Order of Elements

   LinkedHashSet maintains a doubly-linked list running through all of its entries, which defines the iteration order.

   This order is the same as the order in which the elements were added to the LinkedHashSet.

```
public LinkedHashSet(int initialCapacity) {
    super(initialCapacity, .75f, true);
}

public LinkedHashSet() {
    super(16, .75f, true);
}
public LinkedHashSet(Collection<? extends E> c) {
    super(Math.max(2*c.size(), 11), .75f, true);
    addAll(c);
}
```

TreeSet

package java.util;
public class **TreeSet**<E>
extends AbstractSet<E>
implements NavigableSet<E>, Cloneable, java.io.Serializable

1) Null Elements

TreeSet does not permit null elements. Attempting to add a null element will result in a NullPointerException.

2) Constant Time

The size and isEmpty operations run in constant time, O(1).

The contains, add, remove operations run in logarithmic time, O(log n), due to the underlying TreeMap structure that uses a Red-Black tree for maintaining sorted order.

3) Capacity

TreeSet does not have a capacity limit. It dynamically adjusts as elements are added or removed.

The elements are stored in a Red-Black tree structure, which maintains balanced order.

4) Order of Elements

TreeSet maintains elements in a sorted order based on their natural ordering or by a comparator provided at set creation time. The iteration order of the elements is sorted and not based on the insertion order.

private transient NavigableMap<E,Object> m;
private static final Object PRESENT = new Object();

```
public TreeSet() {
    this(new TreeMap<>());
}
TreeSet(NavigableMap<E,Object> m) {
    this.m = m;
}
public TreeSet(Comparator<? super E> comparator) {
    this(new TreeMap<>(comparator));
}


public boolean add(E e) {
    return m.put(e, PRESENT)==null;
}
```

(Map)

Map

package java.util;
public interface **Map**<K,V>

Example:

```
Map<Integer, Person[]> perMap=new HashMap<>();
perMap.put(2,perList);
```

interface Entry<K, V> 键值对类

Map.Entry<String, Integer> provides a way to store and retrieve both the key and value as a single entity.

K getKey();
V getValue(); May return null
V setValue(V value);
public static <K extends Comparable<? super K>, V> Comparator<Map.Entry<K, V>> comparingByKey()
public static <K, V extends Comparable<? super V>> Comparator<Map.Entry<K, V>> comparingByValue()

V put(K key, V value);

V get(Object key);　　根据键获取值 （不存在返回 null）
V remove(Object key);
boolean containsKey(Object key);　　是否包含键
boolean containsValue(Object value);　　是否包含值
default V getOrDefault(Object key, V defaultValue)　　得到 null 时返回默认值

```
public static <K, V extends Comparable<? super V>> Comparator<Map.Entry<K, V>> comparingByValue() {
    return (Comparator<Map.Entry<K, V>> & Serializable)
        (c1, c2) -> c1.getValue().compareTo(c2.getValue());
}
```

int size();
boolean isEmpty();
void clear();
Set<Map.Entry<K, V>> entrySet();

```
    Map.Entry<Integer, Integer> maxEntry = bitCounts.entrySet().stream()
        .max(Map.Entry.comparingByValue())
        .orElse(null); // Returns null if the map is empty
```

Stream

```
        // Create a sample map
        Map<String, Integer> map = new HashMap<>();
        map.put("Alice", 30);
        map.put("Bob", 25);
        map.put("Charlie", 35);
        map.put("David", 20);

        // Sort the map by values
        Map<String, Integer> sortedMap = map.entrySet()
                .stream()
                .sorted(Map.Entry.comparingByValue())
                .collect(Collectors.toMap(
                        Map.Entry::getKey,
                        Map.Entry::getValue,
                        (e1, e2) -> e1, // Merge function in case of duplicate keys
                        LinkedHashMap::new // Preserve the order by using a LinkedHashMap
                ));

        // Print the sorted map
        sortedMap.forEach((key, value) -> System.out.println(key + " -> " + value));
```

default void replaceAll(BiFunction<? super K, ? super V, ? extends V> function)
void clear()　　Removes all of the mappings from this map. The map will be empty after this call returns.

Set<K> keySet();
Collection<V> values();
Returns a Collection view of the values contained in this map.

default V putIfAbsent(K key, V value)
default V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)
The key does not exist initially, the value is null.
Returns the updated value associated with the specified key.

```
map.compute(key, (key, val) -> val != null ? val + 1 : 1)
```

default V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction)
default V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction)

If the value for the specified key is present and non-null, attempts to compute a new mapping given the key and its current mapped value.

```
connectedNodes.computeIfAbsent(node1, k -> new ArrayList<>()).add(node2);
```

default V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V> remappingFunction)

```
map.merge(key, val, (prev, now) -> prev + now)
```

SortedMap

package java.util;

public interface **SortedMap**<K,V> extends Map<K,V>

A Map that further provides a total ordering on its keys.

The map is ordered according to the natural ordering of its keys, or by a Comparator typically provided at sorted map creation time.

Comparator<? super K> comparator();

Returns the comparator used to order the keys in this map, or null if this map uses the natural ordering of its keys.

SortedMap<K,V> subMap(K fromKey, K toKey);

Returns a view of the portion of this map whose keys range from fromKey, inclusive, to toKey, exclusive.

SortedMap<K,V> headMap(K toKey);

Returns a view of the portion of this map whose keys are strictly less than toKey.

SortedMap<K,V> tailMap(K fromKey);

Returns a view of the portion of this map whose keys are greater than or equal to fromKey.

NavigableMap

package java.util;

public interface **NavigableMap**<K,V> extends SortedMap<K,V>

A Map that further provides a total ordering on its keys.

The map is ordered according to the natural ordering of its keys, or by a Comparator typically provided at sorted map creation time.

HashMap

Reference:

https://medium.com/@liberatoreanita/how-hashmap-internally-works-in-java-ad8f054c542#:~:text=Inside%20putVal()%20method%2C%20we,store%20the%20Key%20in%20memory.&text=Hashing%20is%20the%20process%20of,be%20computed%20for%20all%20objects.

https://howtodoinjava.com/java/collections/hashmap/how-hashmap-works-in-java/

package java.util;

public class **HashMap<K, V>**

extends AbstractMap<K, V>

implements Map<K, V>, Cloneable, Serializable

1) Null Elements

   HashMap permits null values and allows one null key.

2) Constant Time

   The size, isEmpty, get, remove, containsKey, getOrDefault, and containsValue operations typically run in constant time, O(1).

   The put operation typically runs in constant time, but in the worst case (e.g., if many elements hash to the same bucket), it can degrade to linear time, O(n).

3) Capacity

As elements are added to a HashMap, its capacity grows automatically. The default initial capacity is 16, and the load factor (default of 0.75) determines when the HashMap needs resizing.
When the number of entries exceeds the product of the load factor and the current capacity, the map is resized (doubled in size by default).

4) Order of Elements
HashMap does not maintain insertion order. The order of elements is based on the hash of the keys and can change when the table is resized.

5) Difference between HashMap and HashTable
Synchronization:
HashMap: It is not synchronized, which means it is not thread-safe. Multiple threads can access and modify a HashMap concurrently without the need for external synchronization.
This makes it faster than Hashtable in a single-threaded environment.
Hashtable: It is synchronized, which means it is thread-safe. Only one thread can modify a Hashtable at a time, which prevents concurrent access issues but can result in slower performance compared to HashMap.
Null Values:
HashMap: Allows one null key and multiple null values.
Hashtable: Does not allow any null key or null value. If you try to store a null key or value, it will throw a NullPointerException.
Iterator:
HashMap: Uses a fail-fast iterator, which means that if the map is modified after the iterator is created (except through the iterator's own remove method), the iterator will throw a ConcurrentModificationException.
Hashtable: Uses an enumerator which is not fail-fast. However, iterators returned by the keySet, entrySet, and values views of the Hashtable are fail-fast.

static final int DEFAULT_INITIAL_CAPACITY = 1 << 4;
static final int MAXIMUM_CAPACITY = 1 << 30;
The maximum capacity, used if a higher value is implicitly specified by either of the constructors with arguments. MUST be a power of two <= 1<<30.

static final float DEFAULT_LOAD_FACTOR = 0.75f;
As the number of elements in the HashMap increases, the capacity is expanded.
The load factor is the measure that decides when to increase the capacity of the Map.
The default load factor is 75% of the capacity.
static final int TREEIFY_THRESHOLD = 8;
The bin count threshold for using a tree rather than list for a bin.
Bins are converted to trees when adding an element to a bin with at least this many nodes.
The value must be greater than 2 and should be at least 8 to mesh with assumptions in tree removal about conversion back to plain bins upon shrinkage.

transient Node<K,V>[] table;
The table, initialized on first use, and resized as necessary. When allocated, length is always a power of two.
(We also tolerate length zero in some operations to allow bootstrapping mechanics that are currently not needed.)
transient int modCount;
int threshold;

The next size value at which to resize (capacity * load factor).

```
static class Node<K,V> implements Map.Entry<K,V>
        final int hash;
        final K key;
        V value;
        Node<K,V> next;
        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }
```

```
public HashMap(int initialCapacity, float loadFactor)
public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}
```

get

```
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(key)) == null ? null : e.value;
}
```

getNode

```
final Node<K,V> getNode(Object key)
```

1) Calculate a hash value (see hash method)
2) Calculate `index i` for the node table to find the bucket.
3) Find the node by checking the node hash value and comparing the key using the equals method.

```
{
    Node<K,V>[] tab; Node<K,V> first, e; int n, hash; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & (hash = hash(key))]) != null) {
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        if ((e = first.next) != null) {
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

put (jdk8 and subsequent)

```
public V put(K key, V value)
```

4) calculate a hash value (see hash method)
5) calculate `index i` for the node table to find the bucket.

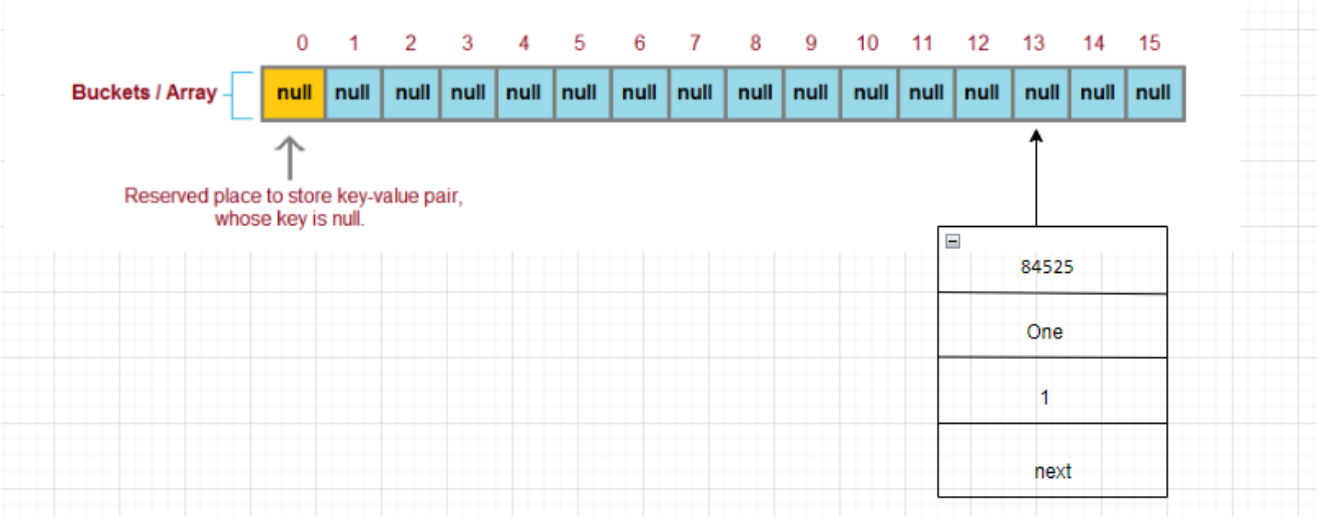
Use a node table with linked list data structure to store data, insert linked list node to the bucket at that index.

At the beginning Node<K,V>[] tab is empty , its initial capacity of the node table is DEFAULT_INITIAL_CAPACITY, and has 0 to 15 buckets.

(The tab variable could be either the original table or the resized table)

```
i = (n - 1) & hash
i = (16 (Default initial capacity) -1) & 84525 (calculate by hash method)
So, i = 13 (in this case)
```

This means that, in index 13 we will have a situation like this:



At index 13 we have first Node that contains following data:

int hash → 84525 calculate by hash method

6) index collision

Insert the new node at the end of node linked list if index collision occurs.

if the length of node linked list reaches TREEIFY_THRESHOLD, treeify the linked list to a Red-Black Tree.

```
{
      return putVal(hash(key), key, value, false, true);
}
```

putVal

```
final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict)
```

Implements Map.put and related methods.

Params:

hash – hash for key

key – the key value – the value to put

onlyIfAbsent – if true, don't change existing value

evict – if false, the table is in creation mode.

Returns:

previous value, or null if none

```
{
    Node<K,V>[] tab; Node<K,V> p; int n, i;
    if ((tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    //A. calculate index i
    // i = (16 (Default initial capacity) -1) & 84525 (calculate by hash method)
    // So, i = 13 (in this case)
    if ((p = tab[i = (n - 1) & hash]) == null)
        tab[i] = newNode(hash, key, value, null);
    else {
        Node<K,V> e; K k;
        if (p.hash == hash &&
```

```
            ((k = p.key) == key || (key != null && key.equals(k))))
            e = p;
        else if (p instanceof TreeNode)
            e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);
        else {
            for (int binCount = 0; ; ++binCount) {
                if ((e = p.next) == null) {
                    p.next = newNode(hash, key, value, null);
                    if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                        treeifyBin(tab, hash);
                    break;
                }
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    break;
                p = e;
            }
        }
        if (e != null) { // existing mapping for key
            V oldValue = e.value;
            if (!onlyIfAbsent || oldValue == null)
                e.value = value;
            afterNodeAccess(e);
            return oldValue;
        }
    }
    ++modCount;
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}
```

hash

```
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

resize

```
final Node<K,V>[] resize()
```

Initializes or doubles table size. If null, allocates in accord with initial capacity target held in field threshold. Otherwise, because we are using power-of-two expansion, the elements from each bin must either stay at same index, or move with a power of two offset in the new table.

Returns:

the table

```
{
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    //A. The table is not empty.
    if (oldCap > 0) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
       //B. The new capacity is twice the original.
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY && oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    //A. The threshold already has a value.
    else if (oldThr > 0)
        newCap = oldThr;
   //A. Initialize threshold and table.
    else {
        newCap = DEFAULT_INITIAL_CAPACITY;
```

```
            newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
        }
        if (newThr == 0) {
            float ft = (float)newCap * loadFactor;
            newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                      (int)ft : Integer.MAX_VALUE);
        }
        threshold = newThr;
        @SuppressWarnings({"rawtypes","unchecked"})
        Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        table = newTab;
        if (oldTab != null) {
            for (int j = 0; j < oldCap; ++j) {
                Node<K,V> e;
                if ((e = oldTab[j]) != null) {
                    oldTab[j] = null;
                    if (e.next == null)
                        newTab[e.hash & (newCap - 1)] = e;
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);
                    else { // preserve order
                        Node<K,V> loHead = null, loTail = null;
                        Node<K,V> hiHead = null, hiTail = null;
                        Node<K,V> next;
                        do {
                            next = e.next;
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)
                                    loHead = e;
                                else
                                    loTail.next = e;
                                loTail = e;
                            }
                            else {
                                if (hiTail == null)
                                    hiHead = e;
                                else
                                    hiTail.next = e;
                                hiTail = e;
                            }
                        } while ((e = next) != null);
                        if (loTail != null) {
                            loTail.next = null;
                            newTab[j] = loHead;
                        }
                        if (hiTail != null) {
                            hiTail.next = null;
                            newTab[j + oldCap] = hiHead;
                        }
                    }
                }
            }
        }
        return newTab;
    }
```

newNode

```
    Node<K,V> newNode(int hash, K key, V value, Node<K,V> next) {
        return new Node<>(hash, key, value, next);
    }
```

Hashtable

package java.util;

public class **Hashtable<K, V>** 线程安全，但效率太低（并发性不如 ConcurrentHashMap，因为 ConcurrentHashMap 引入了分段锁，优先使用 ConcurrentHashMap）

extends Dictionary<K, V>

implements Map<K, V>, Cloneable, Serializable

1) Null Elements

   Hashtable does not permit null keys or null values. Attempting to insert a null key or value will result in a NullPointerException.

2) Constant Time

   The operations size, isEmpty, get, put, remove, containsKey, and containsValue typically run in constant time, O(1).

   However, in the worst case (e.g., if many elements hash to the same bucket), the performance can degrade to linear time, O(n).

3) Capacity

   As elements are added to a Hashtable, its capacity grows automatically.

   However, this growth is constrained by a specified load factor, which determines when to increase the capacity of the hashtable.

   When the number of entries exceeds the product of the load factor and the current capacity, the hashtable is rehashed (i.e., internal data structures are rebuilt).

4) Order of Elements

   Hashtable does not maintain insertion order. The order of elements is based on the hash of the keys and can change when the table is resized or rehashed.

```
private static class Entry<K,V>              键值对象
        implements Map.Entry<K,V>
            final int hash;
            final K key;
            V value;
            Entry<K,V> next;
```

IdentityHashMap

```
package java.util;
public class IdentityHashMap<K,V>
   extends AbstractMap<K,V>
   implements Map<K,V>, java.io.Serializable, Cloneable
```

1) Null Elements

   IdentityHashMap does not permit null keys. If you attempt to insert a null key, it will throw a NullPointerException. However, it does permit null values.

2) Constant Time

   The operations size, isEmpty, get, put, remove, containsKey, and containsValue typically run in constant time, O(1).

   Similar to HashMap, these operations are efficient for retrieving and modifying key-value pairs.

   However, the performance characteristics can degrade in scenarios where hash collisions occur or when there are many entries with similar hash codes, similar to HashMap.

3) Capacity

   Like HashMap, as elements are added to an IdentityHashMap, its capacity grows automatically. It uses a similar mechanism to resize its internal structure to maintain efficiency as more elements are added.

4) Identity-based Equality

   Unlike HashMap, which uses equals() method to check equality of keys, IdentityHashMap uses reference equality (==) for comparing keys.

   This means that two keys are considered equal if and only if they refer to the same object in memory.

public IdentityHashMap()
public IdentityHashMap(int expectedMaxSize)
public IdentityHashMap(Map<? extends K, ? extends V> m)

LinkedHashMap

package java.util;
public class **LinkedHashMap**<K,V>
extends HashMap<K,V>
implements Map<K,V>

1) Null Elements

   LinkedHashMap permits null keys and null values. This means you can store a null value with a specific key, and you can have one entry with a null key.

2) Constant Time

   The operations size, isEmpty, get, put, remove, containsKey, and containsValue typically run in constant time, O(1). These operations are efficient for retrieving and modifying key-value pairs, similar to HashMap.

3) Capacity

   As elements are added to a LinkedHashMap, its capacity grows automatically. It uses a similar mechanism to HashMap to resize its internal structure to maintain efficiency as more elements are added.

4) Order of Elements

   LinkedHashMap maintains the insertion order of elements. The order of elements is based on the order in which keys were inserted into the map. Iterating over the map will return elements in the same order as they were inserted.

Sort By values

```
// Create a sample map
Map<String, Integer> map = new HashMap<>();
map.put("Alice", 30);
map.put("Bob", 25);
map.put("Charlie", 35);
map.put("David", 20);

// Sort the map by values
Map<String, Integer> sortedMap = map.entrySet()
        .stream()
        .sorted(Map.Entry.comparingByValue())
        .collect(Collectors.toMap(
                Map.Entry::getKey,
                Map.Entry::getValue,
                (e1, e2) -> e1, // Merge function in case of duplicate keys
                LinkedHashMap::new // Preserve the order by using a LinkedHashMap
        ));

// Print the sorted map
sortedMap.forEach((key, value) -> System.out.println(key + " -> " + value));
```

public LinkedHashMap(int initialCapacity, float loadFactor)
public LinkedHashMap(int initialCapacity)
public LinkedHashMap()

TreeMap

package java.util;
public class **TreeMap**<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, java.io.Serializable

1) Null Elements

   TreeMap does not permit null keys. If you attempt to insert a null key, it will throw a NullPointerException. However, it does permit null values.

2) Logarithmic Time

   The operations put, get, remove, containsKey, and containsValue typically run in logarithmic time, O(log n).
   This efficiency is achieved because TreeMap uses a Red-Black tree internally to maintain key-value pairs in a sorted order based on the natural ordering of keys or a specified comparator.

3) Capacity

   Unlike HashMap or LinkedHashMap, TreeMap does not have a concept of capacity that grows automatically.
   Instead, it dynamically adjusts the tree structure to maintain balance and efficiency as elements are added or removed.

4) Order of Elements

   TreeMap maintains its elements in sorted order based on the natural ordering of keys (if keys implement Comparable) or using a specified comparator during construction.
   Iterating over a TreeMap will return elements in ascending order of their keys.

private transient int modCount = 0;
private final Comparator<? super K> comparator;

`public TreeMap()`

Constructs a new, empty tree map, using the natural ordering of its keys. All keys inserted into the map must implement the Comparable interface.

```
public TreeMap(Comparator<? super K> comparator)
    // Custom comparator for reverse string order
    Comparator<String> reverseOrder = (s1, s2) -> s2.compareTo(s1);

    // Create TreeMap with the custom comparator
    TreeMap<String, Integer> treeMap = new TreeMap<>(reverseOrder);
```

(Other)

Date

package java.util;
public class **Date**
  implements java.io.Serializable, Cloneable, Comparable<Date>
public Date(long date)
    Date dd = new Date(2014-1900, 6-1, 12); 指定年月日，年份的参数是实际年份减 1900，实际月份减 1。
    Date dd= new Date(19999) 指定当前时间戳（毫秒值，13 位）

public void setTime(long time) 指定当前时间戳（毫秒值）
public long getTime() 获取当前时间戳（毫秒值） // 167 940 0168 936

EventObject

package java.util;

public class **EventObject** implements java.io.Serializable 事件对象
public Object getSource() 事件初始 源对象

EventListener

package java.util;

public interface **EventListener**　事件监听器

Random

package java.util;
public class **Random** implements java.io.Serializable
　　java.util.concurrent.ThreadLocalRandom

public int nextInt()　　返回一个随机整数
public int nextInt(int bound)　取值范围：[0,bound)

ResourceBundle

package java.util;
public abstract class **ResourceBundle**　　xxx.properties 文件操作类

public static final ResourceBundle getBundle(String baseName)　　获取 redis.properties 文件
public final String getString(String key)　　获取键对应的值

Scanner

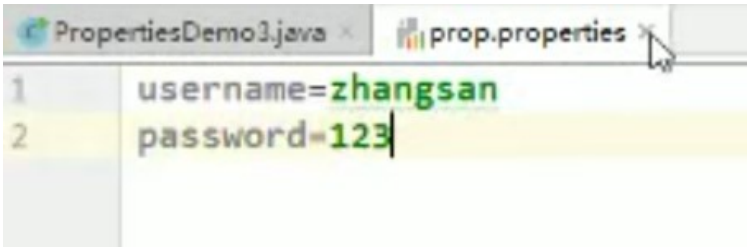
package java.util;
public final class **Scanner** implements Iterator<String>, Closeable　输入类　// Scanner k=new Scanner(System.in);

public Scanner(InputStream source)
public boolean hasNext()　是否有下一个（会阻塞住，直到关闭 scanner）
public boolean hasNextLine()　是否有下一行（会阻塞住，直到关闭 scanner）
public String nextLine()　　返回字符串（回车截至输入，回车换行）
public String next()　　返回字符串（空格截至输入，回车换行）

package java.util;
public interface **Spliterator**<T>　　分流器，和迭代器 Iterator 不同的是，它可以隔断所有元素

Properties

package java.util;
public class **Properties** extends Hashtable<Object,Object>　是一个 Map 体系的集合类



public synchronized void load(InputStream inStream)　　从输入字节流读取属性列表，load 完毕 stream 可以关闭（键和元素对，properties 文件）
public synchronized void load(Reader reader)　　从输入字符流读取属性列表，load 完毕 stream 可以关闭（键和元素对，properties 文件）
public synchronized Object setProperty(String key, String value)　　设置集合的键和值，都是 String 类型，底层调用 Hashtable 方法 put
public String getProperty(String key)　　使用此属性列表中指定的键搜索属性

public Set<String> stringPropertyNames()　　从该属性列表中返回一个不可修改的集，其中健及其对应的值是字符由
public void store(OutputStream out, String comments)　　将此属性列表(键和元素对)写入此 Properties 表中,以适合于使用load(InputStream)方法的格式写入输出字节流
public void store(Writer writer, String comments)　　将此属性列表（键和元素对）写入此 Properties 表中，以适合使用load(Reader)方法的格式写入输出字符流
public synchronized void putAll(Map<?, ?> t)　合并一个 properties 文件或 map 集合

UUID

package java.util;
public final class **UUID**
　implements java.io.Serializable, Comparable<UUID>
public static UUID randomUUID()　　生成一个全局唯一的随机 UUID　　// 569b1416-c38e-4b67-9a47-2ffd303c3f79
uuid.toString().substring(24) = 2ffd303c3f79
public static UUID nameUUIDFromBytes(byte[] name)

(Data Operation)

Arrays

package java.util;
public class **Arrays**

public static <T> Stream<T> stream(T[] array)
　Convert an array to a stream
　Internally, it iterates over all elements once.
　Time Complexity: O(n)
public static <T> List<T> asList(T... a)
　Return an immutable array.
　Immutable Size:
　　The size of the list is fixed and cannot be changed.
　　You cannot add or remove elements from this list, as it will throw an UnsupportedOperationException if you attempt to do so.
　　You can only modify existing elements.
　Arrays.asList(arr)　Convert primitive type array as a single element to a new ArrayList.

public static String toString(Object[] a)
public static String toString(long[] a)
　Print an long array
　// [33, 44, 55]
public static <T> T requireNonNull(T obj, String message)
　If T=null, throw NPE with a message.

public static <T> T[] copyOf(T[] original, int newLength)
　Copy the array, specify the length of the new array, and fill it with 0 if it is not enough
public static <T> T[] copyOfRange(T[] original, int from, int to)
　Copy a specified range of an array into a new array. (e.g. "[from, to)")
　Time Complexity: O(to - from)
　Space Complexity: O(to - from)

public static <T> void setAll(T[] array, IntFunction<? extends T> generator)
　Set all elements of the specified array, using the provided generator function to compute each element.

Time Complexity: O(n)

public static void sort(int[] a)
public static void sort(int[] a, int fromIndex, int toIndex)
public static void sort(long[] a)
public static void sort(short[] a)
public static void sort(byte[] a)
public static void sort(float[] a)
public static void sort(double[] a)
public static void sort(Object[] a)

Time Complexity: O(nlogn)

Space Complexity:

For primitive arrays, consider O(logn) for sorting.

For object arrays, consider O(n) for sorting.

public static <T> void sort(T[] a, Comparator<? super T> c)

Sorts the specified array of objects according to the order induced by the specified comparator.

Arrays.sort method for primitive arrays (like int[]) does not accept a Comparator because primitive types do not support custom comparison logic like objects do.

```
Arrays.sort(envelopes, new Comparator<int[]>() {
    public int compare(int[] e1, int[] e2) {
        if (e1[0] != e2[0]) {
            return e1[0] - e2[0];
        } else {
            return e2[1] - e1[1];
        }
    }
});
Arrays.sort(words, Comparator.comparingInt(String::length));
Arrays.sort(people, (p1, p2) -> p2.name.compareTo(p1.name));
Arrays.sort(points, (p, q) -> p[0] != q[0] ? p[0] - q[0] : q[1] - p[1]);
```

Custom comparator

```
Sort by age, and if ages are equal, sort by name
Comparator<Person> comparator =
       Comparator
       .comparingInt((Person p) -> p.age) // Primary: Sort by age
       .thenComparing(p -> p.name); // Secondary: Sort by name
       // Sort the array
       Arrays.sort(people, comparator);
```

public static void fill(int[] a, int val)

Assigns the specified int value to each element of the specified array of ints.

Time Compleixty: O(n)

public static int binarySearch(long[] a, long key)
public static int binarySearch(int[] a, int key)
public static int binarySearch(int[] a, int fromIndex, int toIndex, int key)

int[] a: The array to be searched.

int fromIndex: The index (inclusive) where the search begins.

int toIndex: The index (exclusive) where the search ends.

int key: The value to search for in the array.

Return Value

If the key is found:

Returns the **index** of the key in the array (within the specified range).

If the key is not found:

Returns a negative value - insertionPoint - 1 , where the **insertionPoint** is the index at which the key would be inserted to maintain the array's sorted order.

```
int[] sortedArr = {1, 3, 5, 7, 9};
int index = Arrays.binarySearch(sortedArr, 4);
System.out.println("Result from binarySearch: " + index); // Output: -3

int[] sortedArr = {1, 3, 3, 3, 5, 7, 9};
int index = Arrays.binarySearch(sortedArr, 3);
System.out.println("Index of 3: " + index); // Output: May return 1, 2, or 3
```

Time Complexity: O(logn)

Search for a specific key in a subrange [fromIndex , toIndex) of a sorted array.

The method requires the array to be sorted only within the specified range [fromIndex, toIndex).

If fromIndex or toIndex is invalid (e.g., out of bounds or fromIndex > toIndex), the method throws ArrayIndexOutOfBoundsException or IllegalArgumentException.

Base64

package java.util;

public class **Base64**

Collections

package java.util;

public class **Collections**

public static final <T> List<T> emptyList() 返回一个空 List，不可修改

public static final <T> Set<T> emptySet() 返回一个空 Set，不可修改

public static <T> List<T> synchronizedList(List<T> list) 获取同步 List，可用于 多个异步任务储存结果

public static boolean isEmpty(Collection coll) 判断数组是否为 null 或者数组长度为 0

public static <T> List<T> synchronizedList(List<T> list) 返回一个线程安全的 list

public static <T extends Comparable<? super T>> void sort(List<T> list)

public static <T> void sort(List<T> list, Comparator<? super T> c)

The sort method modifies the original list in place.

It sorts the elements of the provided list according to the given comparator, and the list itself is rearranged.

// Collections.sort(employees, (o1, o2) -> o1.getName().compareTo(o2.getName()));

public static <T extends Comparable<? super T>> void sort(List<T> list) 排序一个数组

public static void reverse(List<?> list)

Reverse an array

// Collections.reverse(Arrays.asList(a));

public static <T> boolean addAll(Collection<? super T> c, T... elements) 将所有元素添加到一个集合中

static <K,V> Map<K,V> unmodifiableMap(Map<? extends K, ? extends V> m) 只读的 Map，当你调用此 map 的 put 方法时会抛错

public static <T> List<T> unmodifiableList(List<? extends T> list) 只读 List

static class UnmodifiableCollection<E> implements Collection<E>, Serializable 产生只读的 List

public static <E> Set<E> newSetFromMap(Map<E, Boolean> map) 根据 Boolean 的 Map 创建一个 Set

public static <T> void fill(List<? super T> list, T obj)

Replaces all of the elements of the specified list with the specified element.

This method runs in linear time.

Params:

list – the list to be filled with the specified element. obj – The element with which to fill the specified list.

Throws:

UnsupportedOperationException – if the specified list or its list-iterator does not support the set operation.

public static <T> int binarySearch(List<? extends Comparable<? super T>> list, T key)

Performs a binary search on a sorted List.

- It searches for key in the sorted list and returns its index if found.
- If key is not present, it returns -(insertion point) - 1, where insertion point is the index where key should be inserted to maintain order.

```
private static void addSorted(List<Integer> list, int num) {
    int index = Collections.binarySearch(list, num);
    if (index < 0) index = -index - 1; // Convert negative index to insertion point
    list.add(index, num);
}
```

public static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)

Comparator

package java.util;

public interface **Comparator**<T>

A comparison function, which imposes a total ordering on some collection of objects.

Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order.

The item that wins during the comparison will be at the back of the array.

```
int arr[][]=new int[][]{{1},{5},{4}};
Arrays.sort(arr,(a,b)->a[0]-b[0]);
System.out.println(arr[0][0]+" "+arr[1][0]+" "+arr[2][0]);
// 1, 4, 5

int arr[][]=new int[][]{{1},{5},{4}};
Arrays.sort(arr, Comparator.comparingInt(a -> a[0]));
System.out.println(arr[0][0]+" "+arr[1][0]+" "+arr[2][0]);
// 1, 4, 5
```

Use new to create a Comparator:

```
Collections.sort(nodes, new Comparator<int[]>() {
    public int compare(int[] tuple1, int[] tuple2) {
        if (tuple1[0] != tuple2[0]) {
            return tuple1[0] - tuple2[0];
        } else if (tuple1[1] != tuple2[1]) {
            return tuple1[1] - tuple2[1];
        } else {
            return tuple1[2] - tuple2[2];
        }
    }
});
```

public static <T> Comparator<T> nullsFirst(Comparator<? super T> comparator)

```
Collections.sort(employees, Comparator.nullsFirst(Comparator.comparing(Employee::getName)));
```

public static <T extends Comparable<? super T>> Comparator<T> reverseOrder()

public static <T extends Comparable<? super T>> Comparator<T> naturalOrder()

public static <T> Comparator<T> comparingInt(ToIntFunction<? super T> keyExtractor)

Accepts a function that extracts an int sort key from a type T, and returns a Comparator<T> that compares by that sort key.

The returned comparator is serializable if the specified function is also serializable.

Example:

```
Comparator.comparingInt( Person::getAge )
```

Params:

keyExtractor – the function used to extract the integer sort key

public static <T, U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T, ? extends U> keyExtractor)

Accepts a function that extracts a Comparable sort key from a type T, and returns a Comparator<T> that compares by that sort key.

The returned comparator is serializable if the specified function is also serializable.

Example:

```
Comparator<Person> byLastName = Comparator.comparing(Person::getLastName);
```

Params:

keyExtractor – the function used to extract the Comparable sort key

public static <T, U> Comparator<T> comparing(Function<? super T, ? extends U> keyExtractor, Comparator<? super U> keyComparator)

Accepts a function that extracts a sort key from a type T, and returns a Comparator<T> that compares by that sort key using the specified Comparator.

The returned comparator is serializable if the specified function and comparator are both serializable.

Example:

To obtain a Comparator that compares Person objects by their last name ignoring case differences:

```
Comparator<Person> cmp = Comparator.comparing(Person::getLastName, String.CASE_INSENSITIVE_ORDER);
```

Params:

keyExtractor – the function used to extract the sort key keyComparator – the Comparator used to compare the sort key

default <U extends Comparable<? super U>> Comparator<T> thenComparing(Function<? super T, ? extends U> keyExtractor)

Custom comparator 1

```
List<Fruit>listP2=listP.stream().sorted(
   Comparator.comparing(Fruit::getName)
   .reversed()
   .thenComparing(Comparator.comparing(Fruit::getWeight).reversed())
)
.collect(Collectors.toList());
```

Custom comparator 2

```
Sort by age, and if ages are equal, sort by name
Comparator<Person> comparator =
       Comparator
       .comparingInt((Person p) -> p.age) // Primary: Sort by age
       .thenComparing(p -> p.name); // Secondary: Sort by name
       // Sort the array
       Arrays.sort(people, comparator);
```

Specify type explicitly

```
Comparator<int[]> comparator=
    Comparator.comparingInt((int[] point)->point[0]).thenComparing((int[] point)->point[1]);
```

default <U> Comparator<T> thenComparing(Function<? super T, ? extends U> keyExtractor, Comparator<? super U> keyComparator)

Reverse Order

```
Comparator.comparingInt((int[] point) -> point[0])
          .thenComparing((int[] point) -> point[1], Comparator.reverseOrder());
```

int compare(T o1, T o2);

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second.

default Comparator<T> reversed()　　反转　// stream.sorted(Comparator.reverseOrder())

ComparableTimSort

package java.util;

class **ComparableTimSort**

This is a near duplicate of TimSort, modified for use with arrays of objects that implement Comparable, instead of using explicit comparators.

static void sort(Object[] a, int lo, int hi, Object[] work, int workBase, int workLen)

DualPivotQuicksort

package java.util;

final class **DualPivotQuicksort**

static void sort(int[] a, int left, int right,　int[] work, int workBase, int workLen)　排序一个数组　【排序数组，排序左索引，排序右索引　工作数组，工作数组基础可用空间，工作数组可用长度】

Enumeration

package java.util;

public interface **Enumeration**<E>　　枚举（一次获得一个）对象集合中的元素，这种传统接口已被迭代器取代

boolean hasMoreElements()　测试此枚举是否包含更多的元素

E nextElement()　　如果此枚举对象至少还有一个可提供的元素，则返回此枚举的下一个元素

Usage

```
while(iter.hasMoreElements()) {
    String value= iter.nextElement();
}
```

EnumSet

package java.util;

public abstract class **EnumSet**<E extends Enum<E>>

extends AbstractSet<E>

implements Cloneable, java.io.Serializable

public static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)　　创建一个包含指定枚举类里所有枚举值的 EnumSet 集合。

public static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s) 创建一个其元素类型与指定 EnumSet 里元素类型相同的 EnumSet 集合，

新 EnumSet 集合包含原 EnumSet 集合所不包含的、此类枚举类剩下的枚举值（即新 EnumSet 集合和原 EnumSet 集合的集合元素加起来是该枚举类的所有枚举值）。

public static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)　　使用一个普通集合来创建 EnumSet 集合。

public static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)　　创建一个指定 EnumSet 具有相同元素类型、相同集合元素的 EnumSet 集合。

public static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType)　创建一个元素类型为指定枚举类型的空 EnumSet。

public static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)　　创建一个包含一个或多个枚举值的 EnumSet 集合，传入的多个枚举值必须属于同一个枚举类。
public static <E extends Enum<E>> EnumSet<E> range(E from, E to)　　创建一个包含从 from 枚举值到 to 枚举值范围内所有枚举值的 EnumSet 集合。

ImmutableCollections

AAA--SpringBoot.docx#hibernatePersistentBag
AAA--SpringBoot.docx#hibernateCollectionType

package java.util;
class **ImmutableCollections**
static UnsupportedOperationException uoe()
static abstract class AbstractImmutableCollection<E> extends AbstractCollection<E>
public void　clear()　　Throw uoe

Iterator

package java.util;
public interface **Iterator**<E>　迭代器

boolean hasNext();　　是否存在下一个对象元素
E next();　　获取下一个元素（从首个元素开始）
default void remove()　移除元素，不会导致快速失败

ListIterator

package java.util;
public interface **ListIterator**<E> extends Iterator<E>
使用——

```
while(iter.hasNext()) {
    Entry obj = iter.next();        能获得 map 中的每一个键值对了
}
```

Local

package java.util;
public final class **Locale** implements Cloneable, Serializable　　本机操作系统信息，包括语言、国家等信息
public static Locale getDefault(Locale.Category category)　　获取操作系统的本地信息
public String getCountry()　　获取城市　　// CN
public final String getDisplayCountry()　获取显示的城市　// 中国
public String getLanguage()　　获取语言　　// zh
public final String getDisplayLanguage()　获取显示语言　// 中文

```
public enum Category
        DISPLAY("user.language.display",
                "user.script.display",
                "user.country.display",
                "user.variant.display"),
        FORMAT("user.language.format",
               "user.script.format",
               "user.country.format",
               "user.variant.format");
```

Objects

package java.util;
public final class **Objects**　对象工具
public static <T> T requireNonNull(T obj)　对象为空抛出异常，不为空返回对象
public static int hash(Object... values)他　　创建一个 hash 值

Optional

package java.util;
public final class **Optional**<T>

public static <T> Optional<T> of(T value)　　创建一个 Optional 实例，value 不能为空（null 值时，会有空指针异常 NullPointerException）　　// Optional<User> opt = Optional.of(user);
public static <T> Optional<T> ofNullable(T value)　创建一个 Optional 实例，value 可以为空

public void ifPresent(Consumer<? super T> consumer)　　存在则操作，否则不操作　　// ifPresent(user -> log.info(u.getName()));
public T get()　　获取创建创建时的值 value，使用 orThrow 更好，因为 get null 值时有异常
public boolean equals(Object obj)　　判断其他 Optional 内的值是否等于 Optional 的值
public boolean isPresent()　　检查实例是否存在，等同 user != null，不建议使用
public T orElse(T other)　　存在时 直接返回实例，否则返回默认值
public T orElseGet(Supplier<? extends T> other)　　存在时 直接返回实例，否则 通过方法来 设置返回值
public T orNull()　　获取值，不存在返回 null
public <X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier) throws X　　值为空抛 Supplier 继承的异常　　// orElseThrow(() -> new RuntimeException("出错啦"));

public <U> Optional<U> map(Function<? super T, ? extends U> mapper)　　存在使用 map 方法获取 Optional 数据，并返回自动被 Optional 包装的 新数据，　不存在什么都不做 //　map(user -> user.getName())
public <U> Optional<U> flatMap(Function<? super T, Optional<U>> mapper)　使用 flatmap 方法获取可能为 null 的真实数据，并返回 真实值，需要自己手动用 Optional.ofNullable 包装　　// flatmap(user -> user.getName())

使用——

```
List<Integer> list1 = Lists.newArrayList(1, 2, 3);
Optional<List<Integer>> listOptional = Optional.fromNullable(list1);    遍历
listopt.transform(object -> {
    for (int i = 0; i < object.size(); i++) {
        object.set(i, object.get(i) + 1);
    }
    return object;                返回新数组   // Optional.of([2, 3, 4])
}
```

OptionalInt

package java.util;
public final class **OptionalInt**

public int getAsInt()　获取整数值

Spliterator

package java.util;

public interface **Spliterator**<T>　　抽象的能力就是对于一个源头的遍历（traverse）和分区（partition）的能力。
通过 Spliterator 来遍历数据流源头的每个元素（或者一个 bulk 的批量），也通过它来分区数据将其 parallel 并行化。

public static final int ORDERED　= 0x00000010;　　是否有序
public static final int DISTINCT　= 0x00000001;　　是否不同
public static final int IMMUTABLE　= 0x00000400;　　是否不可变

boolean tryAdvance(Consumer<? super T> action);　　单个遍历的能力抽象
default void forEachRemaining(Consumer<? super T> action)　　批量遍历的抽象
Spliterator<T> trySplit();　　分区的抽象

Spliterators

package java.util;

public final class **Spliterators**　分流器工具

public static <T> Spliterator<T> spliterator(Object[] array, int fromIndex, int toIndex, int additionalCharacteristics)　　构造一个分流器

public static <T> Spliterator<T> spliterator(Object[] array, int additionalCharacteristics)　构造一个分流器

public static <T> Spliterator<T> spliterator(Collection<? extends T> c, int characteristics)　构造一个分流器

static class IteratorSpliterator<T> implements Spliterator<T>　　可遍历分流器

static final class DoubleArraySpliterator implements Spliterator.OfDouble　双精度 数组分流器

TimSort

package java.util;

class **TimSort**<T>

static <T> void sort(T[] a, int lo, int hi, Comparator<? super T> c,
T[] work, int workBase, int workLen)

(Time Operation)

Calendar

package java.util;

public abstract class **Calendar** implements Serializable, Cloneable, Comparable<Calendar>

public String getCalendarType()　　The state method of this class.

protected abstract void computeTime();

abstract public void add(int field, int amount);　添加时间信息　　// add(Calendar.DATE, 30);

protected int　　fields[];　　Stores the current calendar time information.

public static Calendar getInstance()　获取当前系统时间实例

public final void set(int year, int month, int date, int hourOfDay, int minute, int second)　　设置当前时间（年月日 时分秒）　// set(2018, 1, 15, 23, 59, 59);

public void set(int field, int value)　　设置当前时间　　// set(Calendar.YEAR, 2018);

public int get(int field)　获取年月日时分秒　　// get(Calendar.MONTH)+1

```
public final void setTime(Date date)        设置时间
public final Date getTime()                 获取 date 对象
public void setTimeInMillis(long millis)    设置时间戳时间
public long getTimeInMillis()               时间戳（毫秒值）
final void internalSet(int field, int value)  Sets the value of given calendar field.
private int compareTo(long t)               Implements the Comparable interface method.
private void updateTime()
```

```
public final static int ERA = 0;
public final static int YEAR = 1;           年份
public final static int MONTH = 2;          月份（从零开始）
public final static int DATE = 5;                   日期     // 11 月 30 日    =>  30
public final static int DAY_OF_MONTH = 5;    日期
public final static int HOUR = 10;          小时
public final static int MINUTE = 12;        分钟
public final static int SECOND = 13;        秒

public final static int WEEK_OF_YEAR = 3;
public final static int WEEK_OF_MONTH = 4;
public final static int DAY_OF_MONTH = 5;
public final static int DAY_OF_YEAR = 6;
public final static int DAY_OF_WEEK = 7;
public final static int DAY_OF_WEEK_IN_MONTH = 8;
```

使用

```
Date date = new Date();//获取当前时间
System.out.println(date);
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");//设置格式

Calendar calendar = Calendar.getInstance(); //创建 Calendar 的实例
calendar.setTime(date);
calendar.add(Calendar.DAY_OF_MONTH, -1); //当前时间减去一天
System.out.println(simpleDateFormat.format(calendar.getTime()));

Calendar calendar2 = Calendar.getInstance();
calendar2.add(Calendar.MONTH, -1);//当前时间减去一个月
System.out.println(simpleDateFormat.format(calendar2.getTime()));

Calendar calendar3 = Calendar.getInstance();
calendar.add(Calendar.YEAR, -1);//当前时间减去一年
System.out.println(simpleDateFormat.format(calendar3.getTime()));
```

TimeZone

```
package java.util;
abstract public class TimeZone implements Serializable, Cloneable
```

GregorianCalendar

package java.util;
public class **GregorianCalendar** extends Calendar
private static final Gregorian gcal = CalendarSystem.getGregorianCalendar();
public GregorianCalendar(TimeZone zone, Locale aLocale)
public String getCalendarType()
protected void computeTime()

JapaneseImperialCalendar

package java.util;
class **JapaneseImperialCalendar** extends Calendar
public String getCalendarType()
protected void computeTime()

(Exceptions)

IlegalFormatConversionException

package java.util;
public class **IllegalFormatConversionException** extends IllegalFormatException String.format()不匹配的参数格式 // String.format("ddd%d","ffff")

UnknownFormatConversionException

package java.util;
public class **UnknownFormatConversionException** extends IllegalFormatException String.format()非法格式 // String.format("ddd%z","ffff")

EmptyStackException

package java.util;
public class **EmptyStackException** extends RuntimeException 空 stack 的 pop()异常

ConcurrentModificationException

package java.util;
public class **ConcurrentModificationException** extends RuntimeException 当我们对集合进行 遍历 的时候，多线程同时对集合进行修改，就会产生并发修改

发生情况：

一边遍历集合，而另一边在修改集合时

在多线程进行插入操作时，由于没有进行同步操作，容易丢失数据。

NoSuchElementException

package java.util;
public class **NoSuchElementException** extends RuntimeException

atomic

AtomicBoolean

package java.util.concurrent.atomic;
public class **AtomicBoolean** implements java.io.Serializable

compareAndSet

```
public final boolean compareAndSet(boolean expectedValue, boolean newValue)
```

Atomically sets the value to newValue if the current value == expectedValue.

Atomic package (values can be updated atomically in a multi-threaded environment. Usually the performance of AtomicInteger is several times that of ReentantLock)

```
{
    return VALUE.compareAndSet(this,
                               (expectedValue ? 1 : 0),
                               (newValue ? 1 : 0));
}
```

AtomicInteger

package java.util.concurrent.atomic;

public class AtomicInteger extends Number implements Serializable

public final int get()　　获取值

public final int getAndIncrement()　　以原子方式将当前值加 1，注意，这里返回的是自增前的值。

public final int incrementAndGet()　　以原子方式将当前值加 1，注意，这里返回的是自增后的值。（getIntVolatile 获取副本值，再调用 compareAndSwapInt 比较旧值和内存值，成功赋值副本值，失败重试）

public final int addAndGet(int data)　　以原子方式将参数与对象中的值相加，并返回结果。

public final int getAndSet(int value)　　以原子方式设置为 newvalue 的值，并返回旧值。

AtomicReference

package java.util.concurrent.atomic;

public class AtomicReference<V> implements Serializable　　将一个对象的所有操作转化成原子操作

LongAdder

package java.util.concurrent.atomic;

public class LongAdder extends Striped64 implements Serializable

The LongAdder class in Java provides a way to perform atomic addition operations on long values with potentially higher throughput than a traditional AtomicLong.

It is particularly useful in scenarios with high contention, where multiple threads are concurrently updating a shared value.

Here's how LongAdder works:

1) Striped Counter

   Internally, LongAdder maintains a set of variables called "cells," which are essentially long counters.

   These cells are striped across to reduce contention.

   When a thread wants to perform an addition operation, it hashes its thread ID to determine which cell it should update.

2) Thread-local Updates

   When a thread wants to increment the value, it first performs the addition operation on its thread-local cell.

   This allows for concurrent updates without contention on a single shared variable.

3) sum() method

   When the value of the LongAdder is needed, such as when calling sum(), the sum is computed by aggregating the values from all the cells,

   along with a base value that is stored directly in the LongAdder object.

   This summing operation ensures correctness even in the presence of concurrent updates.

4) Dynamic Cell Allocation

   If contention is detected and the number of cells is insufficient to mitigate it effectively,

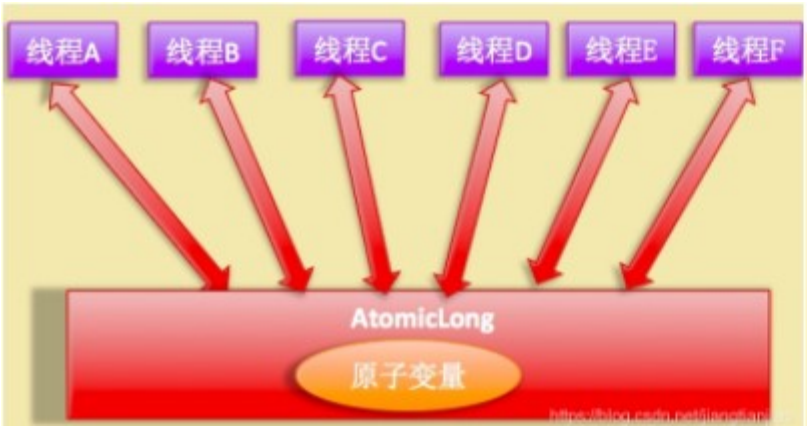
   LongAdder dynamically adds more cells to accommodate the increased contention.

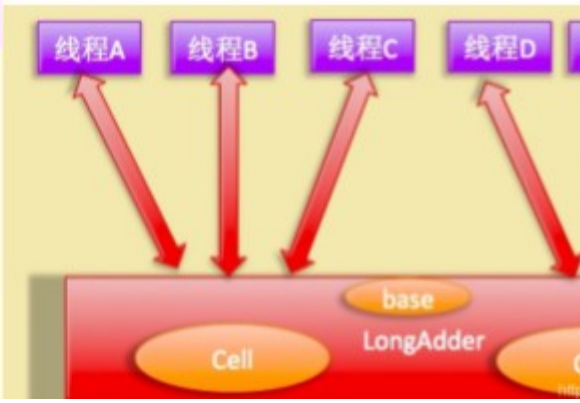
   This dynamic allocation allows LongAdder to adapt to varying levels of contention.

Example:

```
public class ParallelSum {
```

```
    public static void main(String[] args) {
        long[] numbers = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
        LongAdder sum = new LongAdder();

        Arrays.stream(numbers)
              .parallel()
              .forEach(sum::add);

        System.out.println("Sum: " + sum.sum());
    }
}
```





结构：内部维护一个 Cells 数组和一个基值变量 base。数组的下标使用每个线程的 hashcode 值的掩码表示。每个 Cell 里面有一个初始值为 0 的 long 型变量。

数组大小保持 2 的 N 次方大小。

设计目的：减少了高并发情况下对共享资源的争夺。

执行过程：多个线程在争夺同一个原子变量时候，如果失败并不是自旋 CAS 重试，而是尝试获取其他原子变量的锁，最后当获取当前值时候是把所有变量的值累加后再加上 base 的值返回的。

transient volatile long base;

Base value, used mainly when there is no contention, but also as a fallback during table initialization races. Updated via CAS.

transient volatile Cell[] cells;

transient volatile int cellsBusy;

Spinlock (locked via CAS) used when resizing and/or creating Cells.

private static final VarHandle BASE;

private static final VarHandle CELLSBUSY;

private static final VarHandle THREAD_PROBE;

```
final long getAndSetBase(long val) {
    return (long)BASE.getAndSet(this, val);
}
```

```
public void add(long x)
```

The **add** method in the **LongAdder** class in Java is used to atomically increment the current value by a specified amount. Here's how it works:

1) **Thread-local Updates**

When a thread calls the **add(long x)** method on a **LongAdder** instance, it first determines which cell to update based on its thread ID.

Each thread hashes its thread ID to determine the index of the cell it should update.
This allows multiple threads to update different cells concurrently without contention.

2) **Increment Operation**

Once the thread determines the cell to update, it directly increments the value within that cell by the specified amount (**x**).
Since each cell is updated independently by different threads, there is minimal contention.

3) **No Locking**

Unlike traditional atomic variables or synchronized blocks, **LongAdder** does not use locks to protect the counter.
Instead, it relies on the thread-local updates and striped counter mechanism to handle concurrent updates efficiently without contention.

4) **No Blocking**

The **add** method is non-blocking, meaning it does not cause threads to block or wait for each other.
Threads can concurrently update the **LongAdder** without having to wait for each other, leading to better scalability in highly concurrent scenarios.

5) **Aggregation**

When the total sum of the **LongAdder** is needed (e.g., calling **sum()** method), the values from all cells are aggregated,
along with a base value that is stored directly in the **LongAdder** object.
This aggregation operation ensures correctness even in the presence of concurrent updates.

```
{
    Cell[] cs; long b, v; int m; Cell c;
    if ((cs = cells) != null || !casBase(b = base, b + x)) {
        int index = getProbe();
        boolean uncontended = true;
        if (cs == null || (m = cs.length - 1) < 0 ||
            (c = cs[index & m]) == null ||
            !(uncontended = c.cas(v = c.value, v + x)))
            longAccumulate(x, null, uncontended, index);
    }
}
```

sum

```
public long sum() {
    Cell[] cs = cells;
    long sum = base;
    if (cs != null) {
        for (Cell c : cs)
            if (c != null)
                sum += c.value;
    }
    return sum;
}
```

concurrent
(data type)

Delayed

package java.util.concurrent;

public interface **Delayed** extends Comparable<Delayed>

long getDelay(TimeUnit unit);

CopyOnWriteArrayList

package java.util.concurrent;

public class **CopyOnWriteArrayList**<E>

implements List<E>, RandomAccess, Cloneable, java.io.Serializable

CopyOnWriteArrayList is a thread-safe variant of ArrayList in Java. It provides a way to manage concurrent access to a list by multiple threads.

Key Characteristics

Thread-safe

All mutative operations (add, set, remove, etc.) are performed on a copy of the underlying array, ensuring thread safety without the need for external synchronization.

Iterator Safety

Iterators created by CopyOnWriteArrayList are fail-safe and do not throw ConcurrentModificationException. Iterators traverse elements as they were at the time of iterator creation.

Read-Heavy Usage

Ideal for use cases where the list is read frequently but modified infrequently.

Usage

```
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class ConcurrentAccessExample {
    public static void main(String[] args) {
        List<String> list = new CopyOnWriteArrayList<>();
        list.add("Initial");

        ExecutorService executorService = Executors.newFixedThreadPool(3);

        // Task to read from the list
        Runnable readTask = () -> {
            for (String element : list) {
                System.out.println(Thread.currentThread().getName() + " Reading: " + element);
            }
        };

        // Task to modify the list
        Runnable writeTask = () -> {
            list.add("Added by " + Thread.currentThread().getName());
            System.out.println(Thread.currentThread().getName() + " Added an element");
        };

        // Submit read and write tasks
        executorService.submit(readTask);
        executorService.submit(writeTask);
        executorService.submit(readTask);

        // Shutdown the executor service
        executorService.shutdown();
    }
}
```

ConcurrentHashMap

package java.util.concurrent;

public class ConcurrentHashMap<K, V> extends AbstractMap<K, V> implements ConcurrentMap<K, V>, Serializable

Null Elements

ConcurrentHashMap donsn't permits null keys and null values.

结构：由数组数组 Node[]，链表，红黑树组成，结合 CAS 机制+synchronized 同步代码块形式保证线程安全（synchronized1.6 优化了）

存储过程：

根据键的哈希值计算出应存入的索引

如果为 null 则利用 cas 算法将结点添加到数组中

不为 null 则利用 volatile 关键字获得当前位置最新的结点地址，挂在他下面，变成链表。

当链表的长度大于等于 8 时，自动转换成红黑树

多线程：以链表或者红黑树头结点为锁对象，配合悲观锁保证多线程操作集合时数据的安全性。

在默认情况下，最多允许 16 个线程同时访问。

为什么使用红黑树：二叉树在极端情况下会退化成链表，全是左子树或全是右子树，时间复杂度变成 O(n)，而 avl 平衡二叉树，但是任何节点的两个子树的高度差不大于 1，条件太苛刻。

而红黑树折中一点，可以通过左旋右旋来调整树，使之符合红黑树的特征。

```
private static final int DEFAULT_CAPACITY = 16;                Same as HashMap.
private static final int MAXIMUM_CAPACITY = 1 << 30;           Same as HashMap.
private static final float LOAD_FACTOR = 0.75f;                Same as HashMap.
static final int TREEIFY_THRESHOLD = 8;                        Same as HashMap.


transient volatile Node<K,V>[] table;

public ConcurrentHashMap() {}
```

Exchanger

```
package java.util.concurrent;
public class Exchanger<V>
```

ThreadLocalRandom

```
package java.util.concurrent;
@RandomGeneratorProperties(
        name = "ThreadLocalRandom",
        i = 64, j = 0, k = 0,
        equidistribution = 1
)
public class ThreadLocalRandom extends Random
    Thread-Local
        Each thread has its own instance of ThreadLocalRandom.
        This eliminates contention between threads that occurs when they share a common instance of Random.
    High Performance:
        Since each thread has its own random generator, there is no need for synchronization, making it faster in multithreaded
        applications.
    Usage:
        You cannot instantiate ThreadLocalRandom directly using the new keyword.
        Instead, you call ThreadLocalRandom.current() to get the instance specific to the current thread.

public static ThreadLocalRandom current()
```

(thread pool)

Executor

```
package java.util.concurrent;
public interface Executor
void execute(Runnable command);
```

Executors

package java.util.concurrent;

public class Executors

public static ThreadFactory defaultThreadFactory()

newFixedThreadPool

```
public static ExecutorService newFixedThreadPool(int corePoolSize, ThreadFactory threadFactory)
```

Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.

```
{
    return new ThreadPoolExecutor(nThreads, nThreads,
                                  0L, TimeUnit.MILLISECONDS,
                                  new LinkedBlockingQueue<Runnable>());
}
```

newCachedThreadPool

```
public static ExecutorService newCachedThreadPool()
```

Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

Calls to execute will reuse previously constructed threads if available.

If no existing thread is available, a new thread will be created and added to the pool.

```
{
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

newSingleThreadExecutor

```
public static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory)
```

Creates an Executor that uses a single worker thread operating off an unbounded queue.

(Note however that if this single thread terminates due to a failure during execution prior to shutdown, a new one will take its place if needed to execute subsequent tasks.)

Tasks are guaranteed to execute sequentially, and no more than one task will be active at any given time.

Unlike the otherwise equivalent newFixedThreadPool(1) the returned executor is guaranteed not to be reconfigurable to use additional threads.

```
{
    return new FinalizableDelegatedExecutorService
        (new ThreadPoolExecutor(1, 1,
                                0L, TimeUnit.MILLISECONDS,
                                new LinkedBlockingQueue<Runnable>()));
}
```

newScheduledThreadPool

```
public static ScheduledExecutorService newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)
```

Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

```
{
    return new ScheduledThreadPoolExecutor(corePoolSize);
}
```

ExecutorService

package java.util.concurrent;

public interface ExecutorService extends Executor

An Executor that provides methods to manage termination and methods that can produce a Future for tracking progress of one or more asynchronous tasks.

When you create an ExecutorService using Executors.newFixedThreadPool(1),

it creates a thread pool with one thread that remains active even after all tasks are completed.

The JVM will not exit until all non-daemon threads (including the threads in the thread pool) are terminated.

<T> Future<T> submit(Callable<T> task)

Submits a value-returning task for execution and returns a Future representing the pending results of the task.

<T> Future<T> submit(Runnable task, T result);

The Future's get method will return the given result upon successful completion.

void shutdown()

Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

boolean awaitTermination(long timeout, TimeUnit unit) throws InterruptedException;

Blocks until all tasks have completed execution after a shutdown request,
or the timeout occurs, or the current thread is interrupted, whichever happens first.

ThreadPoolExecutor

package java.util.concurrent;

public class ThreadPoolExecutor extends AbstractExecutorService

Thread Selection and Execution:

Idle Threads:

If there are idle threads in the core pool, one of them will pick up the task from the queue and execute it.

Core Pool Size:

If all core threads are busy and the task queue has space, a new thread may be created (if the pool size is less than the core pool size).

Maximum Pool Size:

If the task queue is full and the number of threads is less than the maximum pool size, a new thread may be created to handle the task.

This happens if the task queue type is bounded (like ArrayBlockingQueue).

If the queue is unbounded, it can hold an unlimited number of tasks. The executor will not create any additional threads beyond the corePoolSize,

Queue Full:

If the task queue is full and the number of threads has reached the maximum pool size, the RejectedExecutionHandler is invoked to handle the task.

Thread resource limitation

If it is an unknown number of executions, use a fix-size thread pool(to avoid thread resource consumption) and limit execution time(avoid timeout issues).

If it is a known number of executions, use default thread pool.

public ThreadPoolExecutor(

int corePoolSize,

The number of threads that will always be kept in the pool, even if they are idle.

int maximumPoolSize,

The maximum number of threads that can be created in the pool.

long keepAliveTime,

The amount of time that idle threads will be kept in the pool before they are terminated, The unit of keepAliveTime is specified by the TimeUnit enum

TimeUnit unit,

Specifies the time unit of the duration during which threads may remain idle.

BlockingQueue<Runnable> workQueue,

The queue that will be used to hold tasks that are submitted to the executor

ThreadFactory threadFactory,
RejectedExecutionHandler handler
This parameter specifies how the executor should handle tasks that are rejected because there are no available threads.
)

private static final int RUNNING = -1 << COUNT_BITS;
The thread pool executor is running and accepting new tasks.
This is typically the default state when the thread pool is active and operational.

Calling the shutdown() method of the thread pool can switch to the SHUTDOWN state;
Calling the shutdownNow() method of the thread pool can switch to the STOP state;

private static final int SHUTDOWN = 0 << COUNT_BITS;
The thread pool executor is in the process of shutting down.
It will not accept new tasks but will continue to process tasks that were submitted before the shutdown request.
private static final int STOP = 1 << COUNT_BITS;
The thread pool executor has stopped processing tasks.
It will not accept new tasks and will interrupt any ongoing tasks.
private static final int TIDYING = 2 << COUNT_BITS;
The thread pool executor is in the process of tidying up resources after all tasks have been completed and the pool is shutting down.
After terminated() is executed, it enters the TERMINATED state

private static final int TERMINATED = 3 << COUNT_BITS;
The thread pool executor has completed its shutdown process and is fully terminated.
It will no longer accept new tasks or perform any operations.

private static final RejectedExecutionHandler defaultHandler = new AbortPolicy();

public static class AbortPolicy implements RejectedExecutionHandler
This is the default policy. It throws a RejectedExecutionException when a task is rejected.
public static class DiscardPolicy implements RejectedExecutionHandler
This policy silently discards the task when it is rejected. (no exception will be thrown)
public static class DiscardOldestPolicy implements RejectedExecutionHandler
When a new task is rejected, this policy discards the oldest task in the queue, and insert the new task.
public static class CallerRunsPolicy implements RejectedExecutionHandler
Runs the rejected task in the calling thread.

public int getPoolSize()

public void allowCoreThreadTimeOut(boolean value)
Sets the policy governing whether core threads may time out and terminate if no tasks arrive within the keep-alive time, being replaced if needed when new tasks arrive.
When false, core threads are never terminated due to lack of incoming tasks.
When true, the same keep-alive policy applying to non-core threads applies also to core threads.
To avoid continual thread replacement, the keep-alive time must be greater than zero when setting true.
This method should in general be called before the pool is actively used.

public void shutdown()

Prevents new tasks from being submitted but allows existing tasks to complete.

If you attempt to submit a new task after calling shutdown(), the task will be rejected, and the executor will throw a RejectedExecutionException.

public List<Runnable> shutdownNow()

protected void terminated() { }

public void execute(Runnable command)

Execute a task.

If a Java thread pool is not explicitly shut down after executing tasks, the following can happen:

- Threads Remain Alive

  The threads in the pool will stay alive, waiting for new tasks.

  If the pool is configured with a corePoolSize greater than zero,

  those core threads will persist indefinitely (unless an idle timeout is set for non-core threads).

- JVM Won't Exit

  If the thread pool contains non-daemon threads, the JVM will not shut down automatically because those threads keep running.

  This can lead to a situation where the application appears to hang even after all tasks are completed.

JVM Won't Exit – If the thread pool contains non-daemon threads, the JVM will not shut down automatically because those threads keep running. This can lead to a situation where the application appears to hang even after all tasks are completed.

Thread pool size

For thread pooling, the CPU and task type (CPU-bound vs. I/O-bound) are usually the primary considerations, and disk I/O might not be as directly relevant unless the tasks are heavily dependent on disk operations.

CPU-bound Tasks

Tasks that require heavy computation and are CPU-intensive (e.g., data processing, complex calculations).

Formula:

Thread Pool Size = Number of Cores + 1

Explanation:

If your tasks are primarily CPU-bound (e.g., intensive computations), you should have a thread pool size close to the number of available CPU cores.

Since CPU-bound tasks keep the CPU busy, having more threads than cores will not necessarily improve performance and can lead to context switching overhead.

The "+1" allows a little overhead for handling unexpected delays.

Example1:

For a four-core CPU, the thread pool size would be around 5.

Example2:

Core Pool Size: 4 (matches the number of cores)

Max Pool Size: 4 (keep it equal to the core size for CPU-bound tasks)

Queue Capacity: 0 or a small number (to avoid queuing and keep tasks CPU-bound)

```
@Bean(name = "cpuBoundTaskExecutor")
public TaskExecutor cpuBoundTaskExecutor() {
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(4);
    executor.setMaxPoolSize(4);
    executor.setQueueCapacity(0);
    executor.setThreadNamePrefix("cpu-bound-");
    executor.initialize();
    return executor;
}
```

I/O-bound or Mixed Tasks

Tasks that involve I/O operations such as file reading/writing, network calls, or database interactions.

Formula:

Thread Pool Size = Number of Cores * (1 + Wait Time / Service Time)

Explanation:

For I/O-bound tasks (e.g., tasks involving network calls, disk I/O), where threads spend time waiting for I/O operations to complete,

a larger thread pool can be beneficial. The formula accounts for the time spent waiting (Wait Time) versus the time spent processing (Service Time).

During this time ("I/O wait"), the connection/query/thread is simply "blocked" waiting for the disk.

And it is during this time that the OS could put that CPU resource to better use by executing some more code for another thread.

So, because threads become blocked on I/O, we can actually get more work done by having a number of connections/threads that is greater than the number of physical computing cores.

Example1:

If a task spends 80% of its time waiting (e.g., for I/O) and 20% processing, the formula suggests: Thread Pool Size = 4 * (1 + 0.8/0.2) = 4 * 5 = 20.

Example2:

Core Pool Size: 8-16 (allow more threads to handle the waiting time due to I/O operations)

Max Pool Size: 16-32 (provide additional threads for concurrent I/O operations)

Queue Capacity: 50-100 (can allow queuing since I/O-bound tasks spend time waiting)

```
@Bean(name = "ioBoundTaskExecutor")
public TaskExecutor ioBoundTaskExecutor() {
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(8);
    executor.setMaxPoolSize(32);
    executor.setQueueCapacity(100);
    executor.setThreadNamePrefix("io-bound-");
    executor.initialize();
    return executor;
}
```

General Recommendation:

If you're uncertain about the task profile or the system has mixed workloads, a starting point might be:

CPU-bound: 4 to 6 threads.

I/O-bound: 20 to 40 threads.

Example:

```
@Bean
public TaskExecutor taskExecutor() {
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(4);  // for CPU-bound tasks
    executor.setMaxPoolSize(20);  // for I/O-bound tasks
    executor.setQueueCapacity(50);
    executor.setThreadNamePrefix("my-executor-");
    executor.initialize();
    return executor;
}
```

Example2:

Core Pool Size: 4 (enough to handle moderate concurrent tasks)

Max Pool Size: 8 (provide flexibility to handle occasional bursts)

Queue Capacity: 50-100 (allow some queuing for less critical tasks)

```
@Bean(name = "backgroundTaskExecutor")
public TaskExecutor backgroundTaskExecutor() {
    ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
    executor.setCorePoolSize(4);
```

```
            executor.setMaxPoolSize(8);
            executor.setQueueCapacity(50);
            executor.setThreadNamePrefix("background-task-");
            executor.initialize();
            return executor;
        }
```

Connection pool size

This formula is particularly useful for configuring connection pools to databases or other I/O-bound resources where both CPU and disk I/O are critical factors.

The Formula

connections = ((core_count * 2) + effective_spindle_count)

The "effective spindle count" refers to the number of physical disk spindles (or their equivalent in modern storage technology) that contribute to the I/O performance of the system.

Guess what that means? Your little 4-Core i7 server with one hard disk should be running a connection pool of: 9 = ((4 * 2) + 1). Call it 10 as a nice round number.

Pool-locking

pool size = Tn x (Cm - 1) + 1

Where Tn is the maximum number of threads, and Cm is the maximum number of simultaneous connections held by a single thread.

For example, imagine three threads (Tn=3), each of which requires four connections to perform some task (Cm=4). The pool size required to ensure that deadlock is never possible is:

pool size = 3 x (4 - 1) + 1 = 10

Different thread pools for different types of tasks ensure that a surge in one type of task does not affect others.

ScheduledExecutorService

package java.util.concurrent;

public interface **ScheduledExecutorService** extends ExecutorService

public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit);

Schedule a task to run once after a specified delay

public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit);

Schedule a task to run periodically at a fixed rate

public ScheduledFuture<?> scheduleWithFixedDelay(Runnable command, long initialDelay, long delay, TimeUnit unit);

Schedule a task to run after an initial delay and then repeat at a fixed delay

Inherited Methods

void shutdown();

Shut it down to free up resources

List<Runnable> shutdownNow();

Stop the scheduler immediately

(blocking queue)

【叔祖用上了链子，优先杀死迟到的人，同时找其他人，】

BlockingQueue

package java.util.concurrent;

public interface **BlockingQueue**<E> extends Queue<E>

The difference between blocking queues and regular queues

Blocking Behavior:

Empty Queue:

In a blocking queue, if a thread attempts to retrieve an element and the queue is empty, the thread will be blocked until an element is added to the queue.

In contrast, a regular queue would simply return null or throw an exception.

Full Queue:

In a blocking queue, if a thread attempts to add an element and the queue is full, the thread will be blocked until space becomes available.

In a regular queue, the operation might fail or overwrite an existing element.

Thread Suspension and Wake-Up:

Blocking queues suspend (block) threads when they try to perform operations that cannot be completed immediately due to the queue's state (e.g., retrieving from an empty queue or adding to a full queue).

Once the condition is met (e.g., an element is added to an empty queue or space is freed in a full queue), the blocked thread is automatically woken up to proceed with its operation.

boolean add(E e); Inserts the specified element into this queue

void put(E e) throws InterruptedException; Inserts the specified element into this queue, waiting if necessary for space to become available. (will block the thread)

E take() throws InterruptedException;

boolean offer(E e); Inserts the specified element into this queue

boolean offer(E e, long timeout, TimeUnit unit)

Inserts the specified element into this queue, waiting up to the specified wait time if necessary for space to become available.

E poll(long timeout, TimeUnit unit) throws InterruptedException;

int drainTo(Collection<? super E> c, int maxElements);

int drainTo(Collection<? super E> c);

ArrayBlockingQueue

package java.util.concurrent;

public class ArrayBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, Serializable

A bounded blocking queue backed by an array. This queue orders elements FIFO (first-in-first-out).

Insertion Order:

New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

Capacity:

The capacity is specified at the time of creation and cannot be changed. The array is used to store the elements.

final Object[] items;

int count; Number of elements in the queue

int takeIndex; items index for next take, poll, peek or remove

int putIndex; items index for next put, offer, or add

final ReentrantLock lock;

private final Condition notFull; Condition for waiting takes

private final Condition notEmpty; Condition for waiting puts

public ArrayBlockingQueue(int capacity) Creates an ArrayBlockingQueue with the given (fixed) capacity and default access policy.
public ArrayBlockingQueue(int capacity, boolean fair) Creates an ArrayBlockingQueue with the given (fixed) capacity and the specified access policy.

LinkedBlockingQueue

package java.util.concurrent;
public class LinkedBlockingQueue<E> extends AbstractQueue<E> implements BlockingQueue<E>, java.io.Serializable

An optionally-bounded blocking queue based on linked nodes. This queue orders elements FIFO (first-in-first-out).
Insertion Order
New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.
Linked queues typically have higher throughput than array-based queues but less predictable performance in most concurrent applications.
Capacity
The capacity, if unspecified, is equal to Integer.MAX_VALUE. Linked nodes are dynamically created upon each insertion unless this would bring the queue above capacity.

```
private final int capacity;
private final AtomicInteger count = new AtomicInteger();
transient Node<E> head;
private transient Node<E> last;
private final ReentrantLock takeLock = new ReentrantLock();
private final ReentrantLock putLock = new ReentrantLock();
private final Condition notEmpty = takeLock.newCondition();          Wait queue for waiting takes
private final ReentrantLock putLock = new ReentrantLock();           Lock held by put, offer, etc

static class Node<E> {
  E item;
  Node<E> next;
  Node(E x)
}
private final ReentrantLock takeLock = new ReentrantLock();
private final ReentrantLock putLock = new ReentrantLock();
public LinkedBlockingQueue(int capacity)        Creates a {@code LinkedBlockingQueue} with the given (fixed) capacity.
public LinkedBlockingQueue()                    Creates a LinkedBlockingQueue with a capacity of Integer.MAX_VALUE.
```

LinkedBlockingDeque

package java.util.concurrent;
public class LinkedBlockingDeque<E>

An optionally-bounded blocking deque (double-ended queue) based on linked nodes. This deque orders elements FIFO (first-in-first-out).

Insertion Order:
New elements can be inserted at the head or tail of the deque. Retrieval operations can obtain elements from both ends.
Capacity:

The capacity, if unspecified, is equal to Integer.MAX_VALUE. Linked nodes are dynamically created upon each insertion unless this would bring the deque above capacity.

```
transient Node<E> first;
transient Node<E> last;
private transient int count;            Number of items in the deque
private final int capacity;             Maximum number of items in the deque
final ReentrantLock lock = new ReentrantLock();                 Main lock guarding all access
private final Condition notEmpty = lock.newCondition();         Condition for waiting takes
private final Condition notFull = lock.newCondition();          Condition for waiting puts

static final class Node<E> {
    E item;
    Node<E> prev;
    Node<E> next;
    Node(E x) {
        item = x;
    }
}

public LinkedBlockingDeque()            Creates a LinkedBlockingDeque with a capacity of Integer.MAX_VALUE.
public LinkedBlockingDeque(int capacity)    Creates a LinkedBlockingDeque with the given (fixed) capacity.
void addFirst(E e)
void addLast(E e)
boolean offerFirst(E e)
boolean offerLast(E e)
E peekFirst()
E peekLast()
E take()
E takeFirst()
```

LinkedTransferQueue

```
package java.util.concurrent;
public class LinkedTransferQueue<E> extends AbstractQueue<E>
    implements TransferQueue<E>, java.io.Serializable
```

An unbounded blocking queue based on linked nodes. It is designed for use in producer-consumer queues and supports an extended Transfer interface for synchronous handoffs.

Insertion Order:

New elements are inserted at the tail of the queue, and the queue retrieval operations obtain elements at the head of the queue.

It provides higher throughput for producer-consumer scenarios.

Capacity:

The capacity is unbounded, meaning it can grow as necessary to accommodate additional elements.

```
transient volatile Node head;
private transient volatile Node tail;
```

private transient volatile boolean needSweep; The number of apparent failures to unsplice cancelled nodes

static final class Node implements ForkJoinPool.ManagedBlocker {
  final boolean isData; // false if this is a request node
  volatile Object item; // initially non-null if isData; CASed to match
  volatile Node next;
  volatile Thread waiter; // null when not waiting for a match

public LinkedTransferQueue() Creates an initially empty LinkedTransferQueue.
public LinkedTransferQueue(Collection<? extends E> c) Creates a LinkedTransferQueue initially containing the elements of the given collection, added in traversal order of the collection's iterator.

public void transfer(E e) throws InterruptedException
  Transfers the element to a consumer, waiting if necessary to do so.
  More precisely, transfers the specified element immediately if there exists a consumer already waiting to receive it (in take or timed poll),
  else inserts the specified element at the tail of this queue and waits until the element is received by a consumer.

public boolean tryTransfer(E e)
  Transfers the element to a waiting consumer immediately, if possible.
  More precisely, transfers the specified element immediately if there exists a consumer already waiting to receive it (in take or timed poll),
  otherwise returning false without enqueuing the element.
public boolean tryTransfer(E e, long timeout, TimeUnit unit)
  Transfers the element to a consumer if it is possible to do so before the timeout elapses.
  More precisely, transfers the specified element immediately if there exists a consumer already waiting to receive it (in take or timed poll),
  else inserts the specified element at the tail of this queue and waits until the element is received by a consumer,
  returning false if the specified wait time elapses before the element can be transferred.

public E take() throws InterruptedException
  The consumer in the LinkedTransferQueue class in Java is a thread that takes elements from the queue and processes them.
  The consumer can use the take() method to take an element from the queue, or the poll() method to take an element from the queue if one is available without waiting.

PriorityBlockingQueue

package java.util.concurrent;
public class **PriorityBlockingQueue**<E> extends AbstractQueue<E>
  implements BlockingQueue<E>, java.io.Serializable
  An unbounded blocking queue that uses the same ordering rules as class PriorityQueue and supplies blocking retrieval operations.
  While this queue is logically unbounded, attempted additions may fail due to resource exhaustion (causing OutOfMemoryError).
  This class does not permit null elements.
  A priority queue relying on natural ordering also does not permit insertion of non-comparable objects (doing so results in ClassCastException).

Insertion Order:

Elements are ordered based on their priority. Retrieval operations obtain the element with the highest priority (the least element according to the specified ordering).

Capacity:

The capacity is unbounded, meaning it can grow as necessary to accommodate additional elements.

```
private transient Object[] queue;
private transient int size;
private transient Comparator<? super E> comparator;
private final ReentrantLock lock = new ReentrantLock();
private final Condition notEmpty = lock.newCondition();          Condition for blocking when empty.
private PriorityQueue<E> q;
private transient volatile int allocationSpinLock;               Spinlock for allocation, acquired via CAS.
```

DelayQueue

```
package java.util.concurrent;
public class DelayQueue<E extends Delayed> extends AbstractQueue<E>
  implements BlockingQueue<E>
```

A time-based scheduling queue that orders elements according to a Delay interface.

DelayQueue is a specialized priority queue that holds elements implementing the Delayed interface.

Elements can only be taken from the queue when their delay has expired.

Insertion Order:

Elements are ordered based on the delay expiration. Retrieval operations obtain the element whose delay has expired first.

Capacity:

The capacity is unbounded, meaning it can grow as necessary to accommodate additional elements.

```
private final PriorityQueue<E> q = new PriorityQueue<E>();
```

```
public E take() throws InterruptedException
```

Retrieves and removes the head of this queue, waiting if necessary until an element with an expired delay is available on this queue.

Implementing a Delayed Task

```
import java.util.concurrent.Delayed;
import java.util.concurrent.TimeUnit;

public class DelayedTask implements Delayed {
    private final long delayTime; // delay time in nanoseconds
    private final long startTime; // start time in nanoseconds

    public DelayedTask(long delay, TimeUnit unit) {
        this.delayTime = TimeUnit.NANOSECONDS.convert(delay, unit);
        this.startTime = System.nanoTime() + delayTime;
    }

    @Override
    public long getDelay(TimeUnit unit) {
        long remainingDelay = startTime - System.nanoTime();
        return unit.convert(remainingDelay, TimeUnit.NANOSECONDS);
    }

    @Override
    public int compareTo(Delayed o) {
        if (this.startTime < ((DelayedTask) o).startTime) {
            return -1;
```

```
        }
        if (this.startTime > ((DelayedTask) o).startTime) {
            return 1;
        }
        return 0;
    }

    @Override
    public String toString() {
        return "Task with delay: " + delayTime + " ns";
    }
}
```

Using DelayQueue with Delayed Tasks

```
import java.util.concurrent.DelayQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class DelayQueueExample {
    public static void main(String[] args) throws InterruptedException {
        DelayQueue<DelayedTask> delayQueue = new DelayQueue<>();

        // Adding delayed tasks to the queue
        delayQueue.put(new DelayedTask(5, TimeUnit.SECONDS));
        delayQueue.put(new DelayedTask(10, TimeUnit.SECONDS));
        delayQueue.put(new DelayedTask(15, TimeUnit.SECONDS));

        // Creating a thread pool to process tasks
        ExecutorService executorService = Executors.newFixedThreadPool(1);

        // Processing delayed tasks
        executorService.submit(() -> {
            try {
                while (!delayQueue.isEmpty()) {
                    // Take the task only when its delay has expired
                    DelayedTask task = delayQueue.take();
                    System.out.println("Processing: " + task);
                }
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        });

        // Shutting down the executor service
        executorService.shutdown();
        executorService.awaitTermination(30, TimeUnit.SECONDS);
    }
}
```

SynchronousQueue

package java.util.concurrent;

public class **SynchronousQueue**<E> extends AbstractQueue<E>

implements BlockingQueue<E>, java.io.Serializable

A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa.

A blocking queue in which each insert operation must wait for a corresponding remove operation by another thread, and vice versa.

You cannot peek at a synchronous queue because an element is only present when you try to remove it;
you cannot insert an element (using any method) unless another thread is trying to remove it; you cannot iterate as there is nothing to iterate.

if there is no such queued thread then no element is available for removal and poll() will return null.

Insertion Order:

This queue does not store elements.

Capacity:

A synchronous queue does not have any internal capacity, not even a capacity of one.

public void put(E e) throws InterruptedException Adds the specified element to this queue, waiting if necessary for another thread to receive it.

public E take() throws InterruptedException Retrieves and removes the head of this queue, waiting if necessary for another thread to insert it.

(thread management)

CountDownLatch

【在外面等着，我去里面数一数】

package java.util.concurrent;

public class CountDownLatch

The CountDownLatch class in Java is a synchronization aid that allows one or more threads to wait until a set of operations being performed in other threads completes.

```
private final Sync sync;

public CountDownLatch(int count) {
    if (count < 0) throw new IllegalArgumentException("count < 0");
    this.sync = new Sync(count);
}
```

public void await() throws InterruptedException

Causes the current thread to wait until the latch has counted down to zero, unless the thread is interrupted.

If the current count is zero then this method returns immediately.

public void countDown()

Decrements the count of the latch, releasing all waiting threads if the count reaches zero.

If the current count is greater than zero then it is decremented. If the new count is zero then all waiting threads are re-enabled for thread scheduling purposes.

If the current count equals zero then nothing happens.

Sync

```
private static final class Sync extends AbstractQueuedSynchronizer
{
    private static final long serialVersionUID = 4982264981922014374L;

    Sync(int count) {
        setState(count);
    }

    int getCount() {
        return getState();
    }

    protected int tryAcquireShared(int acquires) {
        return (getState() == 0) ? 1 : -1;
    }

    protected boolean tryReleaseShared(int releases) {
```

```
        // Decrement count; signal when transition to zero
        for (;;) {
            int c = getState();
            if (c == 0)
                return false;
            int nextc = c - 1;
            if (compareAndSetState(c, nextc))
                return nextc == 0;
        }
    }
}
```

Usage

```
import java.util.concurrent.CountDownLatch;

public class CountDownLatchExample {
    public static void main(String[] args) {
        int numberOfWorkers = 3;
        CountDownLatch latch = new CountDownLatch(numberOfWorkers);

        for (int i = 0; i < numberOfWorkers; i++) {
            new Thread(new Worker(latch)).start();
        }

        System.out.println("Main thread is waiting for workers to complete...");

        try {
            latch.await(); // Main thread waits here until the latch count reaches zero.
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        System.out.println("All workers have finished. Main thread proceeds.");
    }
}

class Worker implements Runnable {
    private final CountDownLatch latch;

    public Worker(CountDownLatch latch) {
        this.latch = latch;
    }

    @Override
    public void run() {
        try {
            // Simulate work with sleep
            System.out.println(Thread.currentThread().getName() + " is working...");
            Thread.sleep((long) (Math.random() * 1000));
            System.out.println(Thread.currentThread().getName() + " has finished work.");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            latch.countDown(); // Signal that this thread has completed its work
        }
    }
}
Output:
Main thread is waiting for workers to complete...
Thread-0 is working...
Thread-1 is working...
Thread-2 is working...
Thread-1 has finished work.
Thread-0 has finished work.
Thread-2 has finished work.
All workers have finished. Main thread proceeds.
```

CyclicBarrier

【我在里面多等等就行】

package java.util.concurrent;

public class CyclicBarrier

The CyclicBarrier class in Java is a synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point.

public CyclicBarrier(int parties, Runnable barrierAction)

int await()

Waits until all parties have invoked await on this barrier.

int await(long timeout, TimeUnit unit)

Waits until all parties have invoked await on this barrier, or the specified waiting time elapses.

Usage

```
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class CyclicBarrierExample {
    public static void main(String[] args) {
        int numberOfWorkers = 3;
        CyclicBarrier barrier = new CyclicBarrier(numberOfWorkers, new BarrierAction());

        for (int i = 0; i < numberOfWorkers; i++) {
            new Thread(new Worker(barrier)).start();
        }
    }
}

class Worker implements Runnable {
    private final CyclicBarrier barrier;

    public Worker(CyclicBarrier barrier) {
        this.barrier = barrier;
    }

    @Override
    public void run() {
        try {
            // Perform the first part of the task
            System.out.println(Thread.currentThread().getName() + " is working on the first part...");
            Thread.sleep((long) (Math.random() * 1000));

            // Wait at the barrier
            System.out.println(Thread.currentThread().getName() + " is waiting at the barrier...");
            barrier.await();

            // Perform the second part of the task
            System.out.println(Thread.currentThread().getName() + " is working on the second part...");
            Thread.sleep((long) (Math.random() * 1000));

            // Wait at the barrier again (if needed, demonstrating the cyclic nature)
            System.out.println(Thread.currentThread().getName() + " is waiting at the barrier again...");
            barrier.await();

            System.out.println(Thread.currentThread().getName() + " has finished.");
        } catch (InterruptedException | BrokenBarrierException e) {
            e.printStackTrace();
        }
    }
}

class BarrierAction implements Runnable {
    @Override
```

```
    public void run() {
        System.out.println("All threads have reached the barrier. Performing the barrier action...");
    }
}
Output:
Thread-0 is working on the first part...
Thread-1 is working on the first part...
Thread-2 is working on the first part...
Thread-0 is waiting at the barrier...
Thread-1 is waiting at the barrier...
Thread-2 is waiting at the barrier...
All threads have reached the barrier. Performing the barrier action...
Thread-0 is working on the second part...
Thread-1 is working on the second part...
Thread-2 is working on the second part...
Thread-0 is waiting at the barrier again...
Thread-1 is waiting at the barrier again...
Thread-2 is waiting at the barrier again...
All threads have reached the barrier. Performing the barrier action...
Thread-0 has finished.
Thread-1 has finished.
Thread-2 has finished.
```

Phaser

package java.util.concurrent;

public class **Phaser**

The Phaser class in Java is a flexible synchronization tool that can be used to coordinate multiple threads.

Unlike CountDownLatch or CyclicBarrier, Phaser allows threads to arrive at and wait for a certain phase (step) in the program.

It's especially useful for scenarios where you need to manage multiple phases of execution among multiple threads.

Phases:

A phase represents a step or stage in the program where threads must synchronize.

Parties:

These are the number of threads registered with the Phaser that need to arrive at the synchronization point before the phase can advance.

public Phaser()

public Phaser(int parties)

Usage

```
import java.util.concurrent.Phaser;

public class PhaserExample {
    public static void main(String[] args) {
        // Create a Phaser with no registered parties initially.
        Phaser phaser = new Phaser();

        // Create and start 3 threads (parties)
        for (int i = 0; i < 3; i++) {
            // Register the party with the phaser
            phaser.register();

            new Thread(new Worker(phaser), "Thread-" + i).start();
        }

        // Main thread waiting for phase completion
        // This ensures that all worker threads have completed their tasks before moving to the next phase
        while (phaser.getPhase() < 2) {
            // Wait for all threads to complete the current phase
            phaser.awaitAdvance(phaser.getPhase());
```

```
        }

        System.out.println("All phases are complete!");
    }

    static class Worker implements Runnable {
        private Phaser phaser;

        Worker(Phaser phaser) {
            this.phaser = phaser;
        }

        @Override
        public void run() {
            // Perform phase 0
            System.out.println(Thread.currentThread().getName() + " starting phase 0");
            doWork();
            phaser.arriveAndAwaitAdvance();  // Arrive and wait for others

            // Perform phase 1
            System.out.println(Thread.currentThread().getName() + " starting phase 1");
            doWork();
            phaser.arriveAndAwaitAdvance();  // Arrive and wait for others

            // Perform phase 2
            System.out.println(Thread.currentThread().getName() + " starting phase 2");
            doWork();
            phaser.arriveAndDeregister();    // Arrive and deregister from phaser
        }

        private void doWork() {
            try {
                // Simulate some work with sleep
                Thread.sleep((long) (Math.random() * 1000));
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}
```

Semaphore

package java.util.concurrent;

public class Semaphore implements Serializable

The Semaphore class in Java is a synchronization aid that restricts the number of threads that can access a resource simultaneously.

public Semaphore(int permits)

Creates a Semaphore with the given number of permits and nonfair fairness setting.

Params:

permits – the initial number of permits available. This value may be negative, in which case releases must occur before any acquires will be granted.

public void acquire()

Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted.

Acquires a permit, if one is available and returns immediately, reducing the number of available permits by one.

public void acquire(int permits)

Acquires the given number of permits from this semaphore, blocking until all are available, or the thread is interrupted.

public void release()

Releases a permit, returning it to the semaphore.

public void release(int permits)

Releases the given number of permits, returning them to the semaphore.

public boolean tryAcquire()

Acquires a permit from this semaphore, only if one is available at the time of invocation.

public boolean tryAcquire(long timeout, TimeUnit unit)

Acquires a permit from this semaphore, if one becomes available within the given waiting time and the current thread has not been interrupted.

public boolean tryAcquire(int permits)

Acquires the given number of permits from this semaphore, only if all are available at the time of invocation.

public boolean tryAcquire(int permits, long timeout, TimeUnit unit)

Acquires the given number of permits from this semaphore, if all become available within the given waiting time and the current thread has not been interrupted.

public int availablePermits()

Returns the current number of permits available in this semaphore.

Usage

```
import java.util.concurrent.Semaphore;

public class SemaphoreExample {
    public static void main(String[] args) {
        int numberOfWorkers = 6;
        int permits = 3; // Only 3 threads can access the resource at the same time
        Semaphore semaphore = new Semaphore(permits);

        for (int i = 0; i < numberOfWorkers; i++) {
            new Thread(new Worker(semaphore)).start();
        }
    }
}

class Worker implements Runnable {
    private final Semaphore semaphore;

    public Worker(Semaphore semaphore) {
        this.semaphore = semaphore;
    }

    @Override
    public void run() {
        try {
            System.out.println(Thread.currentThread().getName() + " is waiting for a permit...");
            semaphore.acquire(); // Acquire a permit
            System.out.println(Thread.currentThread().getName() + " acquired a permit.");

            // Simulate work with the shared resource
            System.out.println(Thread.currentThread().getName() + " is working...");
            Thread.sleep((long) (Math.random() * 1000));

            System.out.println(Thread.currentThread().getName() + " has finished work.");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } finally {
            System.out.println(Thread.currentThread().getName() + " releases the permit.");
            semaphore.release(); // Release the permit
        }
    }
}
```

Output:

```
Thread-0 is waiting for a permit...
Thread-1 is waiting for a permit...
Thread-2 is waiting for a permit...
Thread-3 is waiting for a permit...
Thread-4 is waiting for a permit...
Thread-5 is waiting for a permit...
Thread-0 acquired a permit.
Thread-0 is working...
Thread-1 acquired a permit.
Thread-1 is working...
Thread-2 acquired a permit.
Thread-2 is working...
Thread-0 has finished work.
Thread-0 releases the permit.
Thread-3 acquired a permit.
Thread-3 is working...
Thread-1 has finished work.
Thread-1 releases the permit.
Thread-4 acquired a permit.
Thread-4 is working...
Thread-2 has finished work.
Thread-2 releases the permit.
Thread-5 acquired a permit.
Thread-5 is working...
Thread-3 has finished work.
Thread-3 releases the permit.
Thread-4 has finished work.
Thread-4 releases the permit.
Thread-5 has finished work.
Thread-5 releases the permit.
```

(asynchronization)

Flow

package java.util.concurrent;

public final class **Flow**

Future

package java.util.concurrent;

public interface Future<V>

异步编程使用场景：定时服务，拉取结果回写，比如日志，数据库，更新缓存等。

V get() throws InterruptedException, ExecutionException;　阻塞模式

CountedCompleter

package java.util.concurrent;

public abstract class **CountedCompleter**<T> extends ForkJoinTask<T>　触发时执行完成操作，并且没有剩余的待处理操作。 与其他形式的 ForkJoinTasks 相比，CountedCompleters 在子任务停顿和阻塞的情况下通常更强大，但编程不太直观。

CompletionStage

package java.util.concurrent;

public interface **CompletionStage**<T>　　完成阶段动作

public CompletionStage<T> whenComplete (BiConsumer<? super T, ? super Throwable> action)　返回一个新的 CompletionStage

CompletableFuture

package java.util.concurrent;

public class CompletableFuture<T> implements Future<T>, CompletionStage<T>

public static CompletableFuture<Void> allOf(CompletableFuture<?>... cfs)

Wait for multiple CompletableFuture instances to complete.

// CompletableFuture.allOf(allFutures.toArray(new CompletableFuture[0])).join();

// List<ContactDetail> allContactbetails = allFfutures.stream().map(Completableruturet::join)..tolist();

Combine All Results:

```
CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> 1);
CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> 2);
CompletableFuture<Integer> future3 = CompletableFuture.supplyAsync(() -> 3);

CompletableFuture<Void> allOf = CompletableFuture.allOf(future1, future2, future3);

CompletableFuture<Integer> combinedFuture = allOf.thenApply(v -> {
    int result1 = future1.join();
    int result2 = future2.join();
    int result3 = future3.join();
    return result1 + result2 + result3;
});

Integer finalResult = combinedFuture.join(); // Result is 6
System.out.println("Final Result: " + finalResult);
```

public static CompletableFuture<Void> runAsync(Runnable runnable)

Executes a task asynchronously without returning a result.

```
public static void main(String[] args) {
    CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {
        if (true) {
            throw new RuntimeException("Something went wrong!");
        }
    }).exceptionally(ex -> {
        System.out.println("Exception occurred: " + ex.getMessage());
        return null;
    });
```

public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier)

Executes a task asynchronously and returns a result.

The runAsync method uses the ForkJoinPool.commonPool() by default (if no custom executor is provided) to execute the task in a separate thread.

public static <U> CompletableFuture<U> supplyAsync(Supplier<U> supplier, Executor executor)

public static <U> CompletableFuture<U> completedFuture(U value)

Creates a CompletableFuture that is already completed with the given value.

public static <U> CompletableFuture<U> failedFuture(U value)

Returns a new CompletableFuture that is already completed with a given exception.

public <U> CompletableFuture<U> thenCompose(Function<? super T, ? extends CompletionStage<U>> fn)

Chains asynchronous computations by returning a new CompletableFuture.

Use the return result of the previous task as the parameter of the next task

public <U,V> CompletableFuture<V> thenCombine(CompletionStage<? extends U> other, BiFunction<? super T,? super U,? extends V> fn)

Combines the results of two independent CompletableFuture instances and execute the action together when the two are completed together.

BIFunction: the first result and the second result of the combined tasks.

public CompletableFuture<Void> thenAccept(Consumer<? super T> action)

Perform an action with the result of a completed CompletableFuture, without returning a new result.

public CompletableFuture<Void> thenAcceptAsync(Consumer<? super T> action)

Even though both supplyAsync and thenAcceptAsync are using the same ExecutorService (pool), they are different asynchronous tasks, so they are typically executed on different threads from the pool.

public <U> CompletableFuture<U> thenApply(Function<? super T,? extends U> fn)

Transforms the result of the current CompletableFuture into another result using the given function.

If thenApply is used without specifying an Executor, it executes the provided Function on the same thread that completed the previous stage.

public <U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn)

Transforms the result of the current CompletableFuture into another result using the given function, but does so asynchronously.

public <U> CompletableFuture<U> thenApplyAsync(Function<? super T,? extends U> fn, Executor executor)

Even when using the same Executor with the previous stage, thenApplyAsync does not guarantee execution on the same thread as the previous stage.

public CompletableFuture<T> whenComplete(BiConsumer<? super T, ? super Throwable> action)

T The return value of the previous task

Throwable The exception that occurred in the previous task

```
CompletableFuture.supplyAsync(() -> {
    System.out.println("supplyAsync: " + Thread.currentThread().getName());
    return 10;
}).thenApplyAsync(result -> {
    System.out.println("thenApplyAsync: " + Thread.currentThread().getName());
    return result * 2;
}).whenComplete((res, ex) -> {
    System.out.println("whenComplete: " + Thread.currentThread().getName());
});
```

Adds a callback that is executed when the CompletableFuture completes, regardless of success or failure.

whenComplete runs in the same thread that executed thenApplyAsync, regardless of whether an executor is provided.

public <U> CompletableFuture<U> handle(BiFunction<? super T, Throwable, ? extends U> fn)

Provides a way to handle both the result and exception of a CompletableFuture.

public CompletionStage<T> whenCompleteAsync (BiConsumer<? super T, ? super Throwable> action)

Asynchronously executes a callback when the CompletableFuture completes.

public boolean completeExceptionally(Throwable ex)

Completes the CompletableFuture exceptionally with the given throwable (Catching specific exceptions).

public CompletionStage<T> exceptionally (Function<Throwable, ? extends T> fn)

When an exception occurs, if multiple exception handlers are used, only the first one takes effect.

If the computation completes on a thread in the ForkJoinPool, exceptionally typically runs in that same pool but not necessarily on the same thread.

exceptionally() handles exceptions, but its execution thread is not guaranteed to be the same as the runAsync() thread.

public T get() throws InterruptedException, ExecutionException

Retrieves the result of the CompletableFuture. This method blocks until the computation is complete.

Can throw checked exceptions (InterruptedException, ExecutionException). Requires explicit handling of these exceptions.

public T get(long timeout, TimeUnit unit) throws InterruptedException, ExecutionException, TimeoutException

Retrieves the result of the CompletableFuture, with a specified timeout. This method blocks for up to the given time.

public T getNow(T valueIfAbsent)

Retrieves the result of the CompletableFuture if it is already completed, or returns the provided default value if the computation is not yet complete.

public T join()

Retrieves the result of the CompletableFuture. This method blocks until the computation is complete.

Throws CompletionException for exceptional completions, which wraps the actual exception. This makes it easier to handle exceptions in a runtime manner.

Use with a Custom Executor

```
import java.util.concurrent.CompletableFuture;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class CompletableFutureWithThreadPool {
    public static void main(String[] args) {
        ExecutorService threadPool = Executors.newFixedThreadPool(10);

        CompletableFuture<String> future = CompletableFuture.supplyAsync(() -> {
            // Simulate a long-running task
            try {
                Thread.sleep(2000);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
            return "Hello, World!";
        }, threadPool); // Use the custom thread pool

        future.thenAccept(result -> System.out.println("Result: " + result));

        // Shut down the thread pool
        threadPool.shutdown();
    }
}
```

使用————————————

```
// 创建异步执行任务，有返回值
CompletableFuture<Double> cf = CompletableFuture.supplyAsync(()->{
    System.out.println(Thread.currentThread()+" start,time->"+System.currentTimeMillis());
    try {
        Thread.sleep(2000);
    } catch (InterruptedException e) {
    }
    if(true){
        throw new RuntimeException("test");
    }else{
        System.out.println(Thread.currentThread()+" exit,time->"+System.currentTimeMillis());
        return 1.2;
    }
});
System.out.println("main thread start,time->"+System.currentTimeMillis());
//等待子任务执行完成
System.out.println("run result->"+cf.get());
System.out.println("main thread exit,time->"+System.currentTimeMillis());
```

springboot >>

```
@Async("musterExecutor")
public CompletableFuture<Status> execute(Payload payload) {
   HttpHeaders headers = setHeaders();
   String input = Constructor.createPayload(payload);
   HttpEntity<String> requestEntity = new HttpEntity<String>(input, headers);
   Status s = restTemplate.postForObject(execution_url, request_execution, Status.class);
```

```
    return CompletableFuture.completedFuture(s);
}

所有阶段完成再处理>>
static void allOfExample() {
    StringBuilder result = new StringBuilder();
    List messages = Arrays.asList("a", "b", "c");
    List<CompletableFuture> futures = messages.stream()
            .map(msg -> CompletableFuture.completedFuture(msg).thenApply(s -> delayedUpperCase(s)))
            .collect(Collectors.toList());
    CompletableFuture.allOf(futures.toArray(new CompletableFuture[futures.size()])).whenComplete((v, th) -> {
        futures.forEach(cf -> assertTrue(isUpperCase(cf.getNow(null))));
        result.append("done");
    });
    assertTrue("Result was empty", result.length() > 0);
}

public List<R> getAllResult() {
    List<CompletableFuture<R>> futureList = taskList.stream().map(workFunction).collect(Collectors.toList());
    CompletableFuture<Void> allCompletableFuture = CompletableFuture.allOf(futureList.toArray(new
CompletableFuture[futureList.size()]));
    return allCompletableFuture.thenApply(e ->
futureList.stream().map(CompletableFuture::join).collect(Collectors.toList())).join();
}
```

ScheduledExecutorService

package java.util.concurrent;

public interface **ScheduledExecutorService** extends ExecutorService　基于线程池设计的定时任务类,每个调度任务都会分配到线程池中的一个线程去执行,也就是说,任务是并发执行,互不影响

locks

AbstractOwnableSynchronizer

package java.util.concurrent.locks;

public abstract class AbstractOwnableSynchronizer

    implements java.io.Serializable

private transient Thread exclusiveOwnerThread;

AbstractQueuedSynchronizer

package java.util.concurrent.locks;

public abstract class **AbstractQueuedSynchronizer** extends AbstractOwnableSynchronizer implements java.io.Serializable

AbstractQueuedSynchronizer is a utility for creating custom synchronizers in Java, using a node-based queue mechanism.

To customize lock behavior, implement `tryAcquire/tryRelease` for exclusive locks or `tryAcquireShared/tryReleaseShared` for shared locks.

These methods define when a lock can be acquired or released.

The `state` field tracks the lock's status, and `AbstractQueuedSynchronizer` automatically manages the queue and wake-up process for waiting threads.

See: java.util.concurrent.CountDownLatch
java.util.concurrent.ThreadPoolExecutor
java.util.concurrent.locks.ReentrantLock
java.util.concurrent.locks.ReentrantReadWriteLock
java.util.concurrent.Semaphore

State Management:
AQS maintains a single atomic integer state to represent the synchronization state.
The state can be manipulated using methods like getState(), setState(int newState), and compareAndSetState(int expect, int update).

Node-based Queuing:
AQS uses a FIFO queue to manage threads waiting for a lock.
Threads that fail to acquire the lock create a Node and add it to the queue using CAS (Compare-And-Swap).
Each Node stores:
Thread reference (the waiting thread).
State field (e.g., SIGNAL, CANCELLED).
Links to predecessor and successor nodes.
Lock Acquisition & Waiting
A thread checks if the previous node has released the lock (by checking its state).
If the previous node still holds the lock, the current thread blocks using LockSupport.park().
(CLH queue uses spinlocks, but AQS parks threads to avoid busy-waiting).
Lock Release & Node Removal
When a thread releases the lock, it updates the head pointer to the next node.
The previous head node is removed, helping garbage collection.
The next thread in the queue wakes up and acquires the lock.
CLH (Craig, Landin, and Hagersten) Queue Lock
AQS's queue is based on the **CLH lock**, a scalable**, fair spinlock**.
A thread **spins** on its predecessor's lock flag until released.
When the lock is released, the waiting thread acquires **the lock** and removes the old node.

Exclusive and Shared Modes:
AQS supports both exclusive mode and shared mode.
Exclusive mode
Only one thread acquires the resource (e.g., ReentrantLock).
tryAcquire(int arg)
Checks if a single thread can acquire the lock.
tryRelease(int arg)
Updates state to release the lock.
Shared mode
Multiple threads acquire the resource concurrently (e.g., semaphores, ReentrantReadWriteLock).
tryAcquireShared(int arg)
Checks if multiple threads can acquire the resource.
tryReleaseShared(int arg)
Updates state to allow other threads to proceed.

Condition Support:
AQS supports condition variables, which are used to coordinate the suspension and resumption of threads based on certain conditions.
The newCondition() method creates a ConditionObject that can be used with await() and signal() methods.
See: java.util.concurrent.locks.Lock

private transient volatile Node head;

Head of the wait queue, lazily initialized.

private transient volatile Node tail;

Tail of the wait queue. After initialization, modified only via casTail.

private volatile int state;

AQS maintains a volatile int state variable which holds the synchronization state.

Subclasses of AQS define the meaning of this state.

The state variable is manipulated using methods like getState(), setState(int newState), and compareAndSetState(int expect, int update).

These methods ensure that state changes are atomic.

abstract static class Node CLH Nodes

AQS defines an inner static Node class, representing the nodes in the queue. Each node corresponds to a thread waiting to acquire the lock.

```
*  +------+  prev +-------+       +------+
*  | head | <---- | first | <---- | tail |
*  +------+       +-------+       +------+
```

```
volatile Node prev;      // initially attached via casTail
volatile Node next;      // visibly nonnull when signallable
Thread waiter;           // visibly nonnull when enqueued
volatile int status;     // written by owner, atomic bit ops by others

final boolean casPrev(Node c, Node v) {  // for cleanQueue
   return U.weakCompareAndSetReference(this, PREV, c, v);
}
final boolean casNext(Node c, Node v) {  // for cleanQueue
   return U.weakCompareAndSetReference(this, NEXT, c, v);
}
```

public class ConditionObject implements Condition, java.io.Serializable

It provides the mechanism to allow threads to wait for certain conditions to be met and to signal those waiting threads when conditions change.

private transient ConditionNode firstWaiter;

private transient ConditionNode lastWaiter;

public final void await() throws InterruptedException Implements interruptible condition wait.

Causes the current thread to wait until it is signaled or interrupted.

The current thread must hold the lock associated with this ConditionObject.

Releases the lock while waiting and reacquires it before returning.

public final void signal()

Wakes up one waiting thread.

The current thread must hold the lock associated with this ConditionObject.

protected final int getState()

protected final void setState(int newState)

```
protected final boolean compareAndSetState(int expect, int update) {
    return U.compareAndSetInt(this, STATE, expect, update);
}
```

public final void acquire(int arg)
    Acquires in exclusive mode, ignoring interrupts.
    Calls tryAcquire(int arg):
        If true, Lock acquired, no need for queuing.
        If false, The thread joins the queue and blocks until the lock is available.
    Uses addWaiter() and doAcquireInterruptibly() to manage queued threads.
public final boolean release(int arg)
    Releases the lock exclusively and wakes up the next waiting thread if necessary.
protected boolean tryAcquire(int arg)
    Checks if the lock can be acquired in exclusive mode.
    If successful, sets the state accordingly.
protected boolean tryRelease(int arg)
    Updates the state to reflect lock release.

protected int tryAcquireShared(int arg)
protected boolean tryReleaseShared(int arg)

final int acquire(Node node, int arg, boolean shared, boolean interruptible, boolean timed, long time)
public final void acquireInterruptibly(int arg) throws InterruptedException
        Acquires in exclusive mode, aborting if interrupted.

protected boolean isHeldExclusively()
        Returns true if synchronization is held exclusively with respect to the current (calling) thread.
        This method is invoked upon each call to a AbstractQueuedSynchronizer.ConditionObject method.
public final boolean hasQueuedPredecessors()
        Queries whether any threads have been waiting to acquire longer than the current thread.

private static void signalNext(Node h)
public final void acquireShared(int arg)
        Acquires in shared mode, ignoring interrupts.
protected int tryAcquireShared(int arg)
        Tries to acquire in shared mode.
public final boolean releaseShared(int arg)
        Releases in shared mode.

Overrided
protected final void setExclusiveOwnerThread(Thread thread)
        Sets the thread that currently owns exclusive access. A null argument indicates that no thread owns access.
        This method does not otherwise impose any synchronization or volatile field accesses.

acquire
```
public final void acquire(int arg) {
    if (!tryAcquire(arg))
        acquire(null, arg, false, false, false, 0L);
}
```
isEnqueued

```
final boolean isEnqueued(Node node) {
    // Start Traversing from the tail to avoid missing the latest tail node.
    for (Node t = tail; t != null; t = t.prev)
        if (t == node)
            return true;
    return false;
}
```

enqueue

```
final void enqueue(Node node) {
    if (node != null) {
        for (;;) {
            Node t = tail;
            node.setPrevRelaxed(t);        // avoid unnecessary fence
         // If the queue is empty, it initializes the head node.
            if (t == null)
                tryInitializeHead();
         // Adds a node to the tail of the queue.
            else if (casTail(t, node)) {
                t.next = node;
                if (t.status < 0)          // wake up to clean link
                    LockSupport.unpark(node.waiter);
                break;
            }
        }
    }
}
```

CLH Lock Example

```
public class CLHLock {
    private final ThreadLocal<Node> pred;
    private final ThreadLocal<Node> node;
    private final AtomicReference<Node> tail;

    private static class Node {
        volatile boolean locked;
    }

    public CLHLock() {
        this.tail = new AtomicReference<>(new Node());
        this.node = ThreadLocal.withInitial(Node::new);
        this.pred = ThreadLocal.withInitial(() -> null);
    }

    public void lock() {
        final Node node = this.node.get();
        node.locked = true;
        Node pred = this.tail.getAndSet(node);
        this.pred.set(pred);
        while (pred.locked) {
            // Spin-wait
        }
    }

    public void unlock() {
        final Node node = this.node.get();
        node.locked = false;
        this.node.set(this.pred.get());
    }
}
```

Worker

private final class Worker extends AbstractQueuedSynchronizer implements Runnable {

```
    protected boolean tryAcquire(int unused) {
        if (compareAndSetState(0, 1)) {
            setExclusiveOwnerThread(Thread.currentThread());
            return true;
        }
```

```
        return false;
    }

    protected boolean tryRelease(int unused) {
        setExclusiveOwnerThread(null);
        setState(0);
        return true;
    }
}
```

Usage of fair Lock

In a fair lock, we ensure that the threads acquire the lock in the order they requested it. This is typically done by checking if there are any threads in the queue before attempting to acquire the lock.

```
import java.util.concurrent.locks.AbstractQueuedSynchronizer;

public class FairLock {

    private final Sync sync;

    public FairLock() {
        sync = new FairSync();
    }

    private static class FairSync extends AbstractQueuedSynchronizer {
        @Override
        protected boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {
                // Check if the queue is empty or if the current thread is the first in line
                if (!hasQueuedPredecessors() && compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            } else if (current == getExclusiveOwnerThread()) {
                int nextc = c + acquires;
                if (nextc < 0) // overflow
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
                return true;
            }
            return false;
        }

        @Override
        protected boolean tryRelease(int releases) {
            int c = getState() - releases;
            if (Thread.currentThread() != getExclusiveOwnerThread())
                throw new IllegalMonitorStateException();
            boolean free = false;
            if (c == 0) {
                free = true;
                setExclusiveOwnerThread(null);
            }
            setState(c);
            return free;
        }

        @Override
        protected boolean isHeldExclusively() {
            return getState() != 0 && getExclusiveOwnerThread() == Thread.currentThread();
        }
    }

    public void lock() {
        sync.acquire(1);
```

```
    }

    public void unlock() {
        sync.release(1);
    }

    public boolean isLocked() {
        return sync.isHeldExclusively();
    }
}
```

Usage of unfair Lock

An unfair lock allows threads to try to acquire the lock immediately without regard to the order in which they arrived. This can be implemented by ignoring the check for queued threads.

```
import java.util.concurrent.locks.AbstractQueuedSynchronizer;

public class UnfairLock {

    private final Sync sync;

    public UnfairLock() {
        sync = new UnfairSync();
    }

    private static class UnfairSync extends AbstractQueuedSynchronizer {
        @Override
        protected boolean tryAcquire(int acquires) {
            final Thread current = Thread.currentThread();
            int c = getState();
            if (c == 0) {
                if (compareAndSetState(0, acquires)) {
                    setExclusiveOwnerThread(current);
                    return true;
                }
            } else if (current == getExclusiveOwnerThread()) {
                int nextc = c + acquires;
                if (nextc < 0) // overflow
                    throw new Error("Maximum lock count exceeded");
                setState(nextc);
                return true;
            }
            return false;
        }

        @Override
        protected boolean tryRelease(int releases) {
            int c = getState() - releases;
            if (Thread.currentThread() != getExclusiveOwnerThread())
                throw new IllegalMonitorStateException();
            boolean free = false;
            if (c == 0) {
                free = true;
                setExclusiveOwnerThread(null);
            }
            setState(c);
            return free;
        }

        @Override
        protected boolean isHeldExclusively() {
            return getState() != 0 && getExclusiveOwnerThread() == Thread.currentThread();
        }
    }

    public void lock() {
        sync.acquire(1);
```

```
    }

    public void unlock() {
        sync.release(1);
    }

    public boolean isLocked() {
        return sync.isHeldExclusively();
    }
}
```

Usage of newCondition() Method

The Sync class in the example does not have a constructor because it does not require any initial parameters or specific setup beyond what AbstractQueuedSynchronizer (AQS) already provides.

```
import java.util.concurrent.locks.AbstractQueuedSynchronizer;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;

class CustomLock implements Lock {
    private final Sync sync = new Sync();

    // Custom synchronizer extending AQS
    private static class Sync extends AbstractQueuedSynchronizer {
        @Override
        protected boolean tryAcquire(int arg) {
            if (compareAndSetState(0, 1)) {  // If state is 0 (unlocked), set it to 1 (locked)
                setExclusiveOwnerThread(Thread.currentThread());
                return true;
            }
            return false;
        }

        @Override
        protected boolean tryRelease(int arg) {
            if (getState() == 0) throw new IllegalMonitorStateException();
            setExclusiveOwnerThread(null);
            setState(0);
            return true;
        }

        @Override
        protected boolean isHeldExclusively() {
            return getState() == 1 && getExclusiveOwnerThread() == Thread.currentThread();
        }

        // Create a ConditionObject using newCondition()
        Condition newCondition() {
            return new ConditionObject();
        }
    }

    @Override
    public void lock() {
        sync.acquire(1);
    }

    @Override
    public void unlock() {
        sync.release(1);
    }

    @Override
    public Condition newCondition() {
        return sync.newCondition();
    }
```

```
    @Override
    public boolean tryLock() {
        return sync.tryAcquire(1);
    }

    @Override
    public void lockInterruptibly() throws InterruptedException {
        sync.acquireInterruptibly(1);
    }

    @Override
    public boolean tryLock(long time, java.util.concurrent.TimeUnit unit) throws InterruptedException {
        return sync.tryAcquireNanos(1, unit.toNanos(time));
    }
}
```

Lock

package java.util.concurrent.locks;

public interface **Lock**

It provides a mechanism for thread synchronization that is more versatile than synchronized blocks.

Advantages Over Synchronized Blocks:

Fairness**:**

Lock:

Can be configured to use a fair locking policy, ensuring that the longest-waiting thread acquires the lock next.

Synchronized:

Uses an internal, non-fair locking policy. Threads are not guaranteed to acquire the lock in the order they requested it.

Interruptibility:

Lock:

Allows threads to be interrupted while waiting for a lock. This can be achieved using the lockInterruptibly() method.

Synchronized:

Does not support interruptible locking; threads waiting for a lock cannot be interrupted.

Try-Locking:

Lock:

Provides methods like tryLock() to attempt to acquire the lock without blocking. This can be useful for avoiding deadlocks or for implementing non-blocking algorithms.

Synchronized:

Does not have a non-blocking lock acquisition mechanism.

Timed Locking:

Lock:

Supports timed locking with tryLock(long time, TimeUnit unit), allowing a thread to attempt to acquire the lock for a specific period.

Synchronized:

Does not provide built-in support for timed locking.

Multiple Condition Variables:

Lock:

Can create multiple Condition objects using newCondition(), providing a more flexible way to handle multiple conditions within the same lock.

Synchronized:

Provides a single intrinsic monitor for condition handling, which limits flexibility.

Locking Scope and Order:

Lock:

Allows locking and unlocking to be done in different scopes, which can help with complex locking strategies. You can acquire and release locks in different orders.

Synchronized:

The scope is confined to the block or method, and locks are released automatically when exiting the block or method, which can be less flexible for complex scenarios.

void lock();
void unlock();

boolean tryLock();
Condition newCondition();

Condition

package java.util.concurrent.locks;
public interface Condition

Condition and Object

The awiat method of the Condition class is equivalent to the wait method of the Object class
The signal method of the Condition class is equivalent to the notify method of the Object class
The signalAll method of the Condition class is equivalent to the notifyAll method of the Object class

The ReentrantLock class can wake up threads with specified conditions, while the wakeup of the object is random.

void await() throws InterruptedException;

Causes the current thread to wait until it is signaled or interrupted.
The thread releases the associated lock and enters the waiting state.
When the thread is signaled, it must reacquire the lock before it can proceed.

boolean await(long time, TimeUnit unit) throws InterruptedException;

Causes the current thread to wait for the specified amount of time (in the given time unit) or until it is signaled.
Returns true if the thread was signaled before the timeout, and false if the timeout occurred.

void awaitUninterruptibly();

Similar to await(), but the thread cannot be interrupted while waiting. It only resumes when it is signaled.

long awaitNanos(long nanosTimeout) throws InterruptedException;

Causes the current thread to wait for the specified amount of time (in nanoseconds) or until it is signaled.
Returns the remaining time in nanoseconds if the condition is signaled before the timeout.

boolean awaitUntil(Date deadline) throws InterruptedException;

Causes the current thread to wait until the given deadline or until it is signaled.
Returns true if the thread was signaled before the deadline, and false if the deadline was reached.

void signal();

Wakes up one waiting thread (chosen arbitrarily) that is waiting on this Condition. The awakened thread must reacquire the lock before it can proceed.
This is similar to notify() in the traditional monitor model.

void signalAll();

Wakes up all the threads that are waiting on this Condition. All awakened threads will compete to reacquire the lock.
This is similar to notifyAll() in the traditional monitor model.

Lock

package java.util.concurrent.locks;

public interface **Lock**

void lockInterruptibly() throws InterruptedException;

Condition newCondition();

boolean tryLock();

boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

Usage

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.concurrent.locks.Condition;
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class ProducerConsumer {
    private final Queue<Integer> queue = new LinkedList<>();
    private final int MAX_CAPACITY = 5;
    private final Lock lock = new ReentrantLock();
    private final Condition notEmpty = lock.newCondition();
    private final Condition notFull = lock.newCondition();

    public void produce(int value) throws InterruptedException {
        lock.lock();
        try {
            while (queue.size() == MAX_CAPACITY) {
                notFull.await(); // Wait until space is available
            }
            queue.offer(value);
            System.out.println("Produced: " + value);
            notEmpty.signal(); // Notify consumers that items are available
        } finally {
            lock.unlock();
        }
    }

    public void consume() throws InterruptedException {
        lock.lock();
        try {
            while (queue.isEmpty()) {
                notEmpty.await(); // Wait until items are available
            }
            int value = queue.poll();
            System.out.println("Consumed: " + value);
            notFull.signal(); // Notify producers that space is available
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        ProducerConsumer pc = new ProducerConsumer();

        // Producer thread
        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    pc.produce(i);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        }).start();
```

```
        // Consumer thread
        new Thread(() -> {
            for (int i = 0; i < 10; i++) {
                try {
                    pc.consume();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        }).start();
    }
}
```

ReentrantLock

package java.util.concurrent.locks;

public class ReentrantLock implements Lock, Serializable

Key Features of ReentrantLock

- Reentrancy

  The lock is reentrant, meaning that the thread that holds the lock can reacquire it without deadlock.
  This is useful when a thread enters a lock-protected section of code, and then calls another method that is also lock-protected.
  Use the state field to check whether the lock is acquired or how many times it has been reentered.

- Lock Fairness

  ReentrantLock can be configured to be fair or non-fair. A fair lock favors granting access to the longest-waiting thread, whereas a non-fair lock does not guarantee any particular access order.

- Condition Variables

  ReentrantLock provides support for multiple condition variables through its newCondition() method.
  Conditions allow threads to wait for specific conditions to be met before continuing execution,
  which is more flexible than the intrinsic condition mechanism provided by synchronized blocks and wait()/notify() methods.

- Interruptible Lock Acquisition:

  Lock acquisition with ReentrantLock can be interruptible, meaning that a thread attempting to acquire a lock can be interrupted.

- Lock Polling

  ReentrantLock provides methods for attempting to acquire the lock without blocking indefinitely (tryLock() methods), allowing for more responsive and flexible concurrency control.

What is the difference between "ReentrantLock" and "synchronized keyword" in java?

Ease of Use:

- synchronized is easier to use and less error-prone for simple synchronization needs.
- ReentrantLock is more flexible and powerful, but requires explicit locking and unlocking, which can be error-prone if not managed correctly.

Fairness:

- synchronized does not provide fairness.
- ReentrantLock can be configured to be fair, ensuring fair access among competing threads.

Condition Variables:

- synchronized uses intrinsic condition variables via wait(), notify(), and notifyAll().
- ReentrantLock uses explicit condition variables via newCondition(), allowing multiple conditions per lock.

Try and Timed Locking:

- synchronized does not support try-locking or timed-locking.
- ReentrantLock supports non-blocking try-locking and timed-locking.

public ReentrantLock()
public ReentrantLock(boolean fair)

private final Sync sync;

public Condition newCondition()
final int getQueueLength()
int getWaitQueueLength(Condition condition)
boolean hasWaiters(Condition condition)
final boolean hasQueuedThread(Thread thread)
final boolean hasQueuedThreads()
final boolean isFair()
boolean isHeldByCurrentThread()
boolean isLocked()

void lockInterruptibly()
Acquires the lock unless the current thread is interrupted.
Acquires the lock if it is not held by another thread and returns immediately, setting the lock hold count to one.
If the current thread already holds this lock then the hold count is incremented by one and the method returns immediately.
If the lock is held by another thread then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of two things happens:
The lock is acquired by the current thread; or
Some other thread interrupts the current thread.

```
public void lock()
```

Acquires the lock if it is not held by another thread and returns immediately, setting the lock hold count to one.
If the current thread already holds the lock then the hold count is incremented by one and the method returns immediately.
If the lock is held by another thread then the current thread becomes disabled for thread scheduling purposes and lies dormant until the lock has been acquired,
at which time the lock hold count is set to one.

```
{
    sync.lock();
}
public void unlock()
```

Attempts to release this lock.
If the current thread is the holder of this lock then the hold count is decremented.
If the hold count is now zero then the lock is released. If the current thread is not the holder of this lock then IllegalMonitorStateException is thrown.
Throws:
IllegalMonitorStateException – if the current thread does not hold this lock

```
{
    sync.release(1);
}
```

boolean tryLock()
boolean tryLock(long timeout, TimeUnit unit)
final int getHoldCount()

hasQueuedPredecessors
public final boolean hasQueuedPredecessors()

```
     Returns:
          true if there is a queued thread preceding the current thread, and false if the current thread is at the head of the queue
          or the queue is empty
{
   Thread first = null; Node h, s;
   if ((h = head) != null && ((s = h.next) == null ||
                        (first = s.waiter) == null ||
                        s.prev == null))
      first = getFirstQueuedThread(); // retry via getFirstQueuedThread
   return first != null && first != Thread.currentThread();
}
getFirstQueuedThread
public final Thread getFirstQueuedThread() {
   Thread first = null, w; Node h, s;
   if ((h = head) != null && ((s = h.next) == null ||
                        (first = s.waiter) == null ||
                        s.prev == null)) {
      // traverse from tail on stale reads
      for (Node p = tail, q; p != null && (q = p.prev) != null; p = q)
         if ((w = p.waiter) != null)
            first = w;
   }
   return first;
}
Sync
abstract static class Sync extends AbstractQueuedSynchronizer
          final boolean tryLock() {
             Thread current = Thread.currentThread();
             int c = getState();
             if (c == 0) {
                if (compareAndSetState(0, 1)) {
                   setExclusiveOwnerThread(current);
                   return true;
                }
             } else if (getExclusiveOwnerThread() == current) {
                if (++c < 0) // overflow
                   throw new Error("Maximum lock count exceeded");
                setState(c);
                return true;
             }
             return false;
          }

          @ReservedStackAccess
          protected final boolean tryRelease(int releases) {
             int c = getState() - releases;
             if (getExclusiveOwnerThread() != Thread.currentThread())
                throw new IllegalMonitorStateException();
```

```
            boolean free = (c == 0);
            if (free)
                setExclusiveOwnerThread(null);
            setState(c);
            return free;
        }

        final ConditionObject newCondition() {
            return new ConditionObject();
        }
```

FairSync

static final class FairSync extends Sync

```
    protected final boolean tryAcquire(int acquires) {
        if (getState() == 0 && !hasQueuedPredecessors() &&
            compareAndSetState(0, acquires)) {
            setExclusiveOwnerThread(Thread.currentThread());
            return true;
        }
        return false;
    }
```

Usage

```
import java.util.concurrent.locks.ReentrantLock;

public class SafeCounter {
    private final ReentrantLock lock = new ReentrantLock();
    private int count = 0;

    // Method to increment the counter
    public void increment() {
        lock.lock(); // Acquire the lock
        try {
            count++; // Critical section
        } finally {
            lock.unlock(); // Always release the lock in a finally block
        }
    }

    // Method to get the current counter value
    public int getCount() {
        lock.lock(); // Acquire the lock
        try {
            return count; // Critical section
        } finally {
            lock.unlock(); // Always release the lock in a finally block
        }
    }

    public static void main(String[] args) {
        SafeCounter counter = new SafeCounter();

        // Create multiple threads that increment the counter
        Thread[] threads = new Thread[10];
        for (int i = 0; i < 10; i++) {
            threads[i] = new Thread(() -> {
                for (int j = 0; j < 1000; j++) {
                    counter.increment();
                }
            });
        }

        // Start all threads
```

```
        for (Thread thread : threads) {
            thread.start();
        }

        // Wait for all threads to finish
        for (Thread thread : threads) {
            try {
                thread.join();
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }

        // Print the final count value
        System.out.println("Final count: " + counter.getCount());
    }
}
```

使用——

MAIN >>

```
private Lock lock = new ReentrantLock();
//Lock lock=new ReentrantLock(true);//公平锁
//Lock lock=new ReentrantLock(false);//非公平锁
private Condition condition=lock.newCondition();//创建 Condition
public void testMethod() {
    try {
        lock.lock();        //lock 加锁
        condition.await();   //开始 wait 方法等待（通过创建 Condition 对象来使线程 wait，必须先执行 lock.lock 方法获得锁）
        condition.signal();  //signal 方法唤醒（condition 对象的 signal 方法可以唤醒 wait 线程）
        for (int i = 0; i < 5; i++) {
            System.out.println("ThreadName=" + Thread.currentThread().getName()+ (" " + (i + 1)));
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    finally{
        lock.unlock();
    }
}
```

ReadWriteLock

package java.util.concurrent.locks;

public interface **ReadWriteLock**　读写锁，在读的地方使用读锁，在写的地方使用写锁，如 果没有写锁的情况下，读是无阻塞的（多个读锁不互斥，读锁与写锁互斥，这是由 jvm 自己控制的，你只要上好相应的锁即可）

ReentrantReadWriteLock

package java.util.concurrent.locks;

public class **ReentrantReadWriteLock**

implements ReadWriteLock, java.io.Serializable

Read and Write Locks:

The ReentrantReadWriteLock maintains a pair of locks: a read lock and a write lock.

The read lock can be held simultaneously by multiple threads as long as there are no writers.

The write lock is exclusive and allows only one thread to modify the shared resource at a time.

Reentrancy:

Both the read lock and the write lock are reentrant. A thread that holds a read lock or write lock can reacquire the same lock without blocking.

Lock Downgrading:

A thread holding the write lock can acquire the read lock without releasing the write lock, allowing for "lock downgrading".

However, upgrading from a read lock to a write lock is not allowed.

Fair and Non-Fair Modes:

Like ReentrantLock, ReentrantReadWriteLock can be created in fair mode, where access is granted in the order in which the requests are made, or non-fair mode, where the order is not guaranteed.

State Encoding

In a read-write lock implementation, the state is usually an integer that encodes both the count of shared locks and the presence of an exclusive lock.

The SHARED_SHIFT helps to position the shared lock count in the integer, and the EXCLUSIVE_MASK helps to isolate the exclusive lock count.

```
private final ReentrantReadWriteLock.ReadLock readerLock;
private final ReentrantReadWriteLock.WriteLock writerLock;
final Sync sync;
```

Sync

```
abstract static class Sync extends AbstractQueuedSynchronizer
    static final int SHARED_SHIFT   = 16;
    static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;

    static final int SHARED_UNIT= (1 << SHARED_SHIFT);
    static final int EXCLUSIVE_MASK = (1 << SHARED_SHIFT) - 1;

    private transient ThreadLocalHoldCounter readHolds;
    Sync() {
         readHolds = new ThreadLocalHoldCounter();
         setState(getState()); // ensures visibility of readHolds
    }
    static int sharedCount(int c)    { return c >>> SHARED_SHIFT; }
    static int exclusiveCount(int c) { return c & EXCLUSIVE_MASK; }
```

```
    @ReservedStackAccess
    protected final boolean tryRelease(int releases) {
        if (!isHeldExclusively())
            throw new IllegalMonitorStateException();
        int nextc = getState() - releases;
        boolean free = exclusiveCount(nextc) == 0;
        if (free)
            setExclusiveOwnerThread(null);
        setState(nextc);
        return free;
    }

    @ReservedStackAccess
    protected final boolean tryAcquire(int acquires)
```

Walkthrough:

1. If read count nonzero or write count nonzero and owner is a different thread, fail.
2. If count would saturate, fail. (This can only happen if count is already nonzero.)

3. Otherwise, this thread is eligible for lock if it is either a reentrant acquire or queue policy allows it. If so, update state and set owner.

```
    {
        Thread current = Thread.currentThread();
        int c = getState();
        int w = exclusiveCount(c);
        if (c != 0) {
            // (Note: if c != 0 and w == 0 then shared count != 0)
            if (w == 0 || current != getExclusiveOwnerThread())
                return false;
            if (w + exclusiveCount(acquires) > MAX_COUNT)
                throw new Error("Maximum lock count exceeded");
            // Reentrant acquire
            setState(c + acquires);
            return true;
        }
        if (writerShouldBlock() ||
            !compareAndSetState(c, c + acquires))
            return false;
        setExclusiveOwnerThread(current);
        return true;
    }
```

Usage

```
import java.util.concurrent.locks.ReentrantReadWriteLock;

public class ReadWriteCounter {
    private final ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
    private final ReentrantReadWriteLock.ReadLock readLock = rwLock.readLock();
    private final ReentrantReadWriteLock.WriteLock writeLock = rwLock.writeLock();
    private int count = 0;

    // Method to increment the counter
    public void increment() {
        writeLock.lock();
        try {
            count++;
        } finally {
            writeLock.unlock();
        }
    }

    // Method to get the current counter value
    public int getCount() {
        readLock.lock();
        try {
            return count;
        } finally {
            readLock.unlock();
        }
    }

    public static void main(String[] args) {
        ReadWriteCounter counter = new ReadWriteCounter();

        // Create multiple reader threads
        Thread[] readers = new Thread[5];
        for (int i = 0; i < 5; i++) {
            readers[i] = new Thread(() -> {
                for (int j = 0; j < 10; j++) {
                    System.out.println(Thread.currentThread().getName() + " reads count: " +
counter.getCount());
                    try {
                        Thread.sleep(100);
                    } catch (InterruptedException e) {
```

```
                    Thread.currentThread().interrupt();
                }
            }
        });
    }

    // Create multiple writer threads
    Thread[] writers = new Thread[2];
    for (int i = 0; i < 2; i++) {
        writers[i] = new Thread(() -> {
            for (int j = 0; j < 5; j++) {
                counter.increment();
                System.out.println(Thread.currentThread().getName() + " increments count");
                try {
                    Thread.sleep(200);
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                }
            }
        });
    }

    // Start all reader and writer threads
    for (Thread reader : readers) {
        reader.start();
    }
    for (Thread writer : writers) {
        writer.start();
    }

    // Wait for all threads to finish
    try {
        for (Thread reader : readers) {
            reader.join();
        }
        for (Thread writer : writers) {
            writer.join();
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }

    // Print the final count value
    System.out.println("Final count: " + counter.getCount());
  }
}
```

StampedLock

package java.util.concurrent.locks;

public class **StampedLock** implements java.io.Serializable

Optimistic Read Lock:

StampedLock provides an optimistic read lock, which allows multiple threads to read a shared resource concurrently without blocking.

Unlike a traditional read lock, the optimistic read lock doesn't acquire a true lock and thus doesn't block any other operations.

However, after reading the shared resource, the thread must validate that no write operation occurred during the read (using a stamp).

Read Lock:

When a thread needs to ensure that no write operations occur during its read, it can acquire a traditional read lock.

This lock is shared among multiple readers but will block if a write lock is requested.

Write Lock:

For exclusive access to a shared resource, a thread can acquire a write lock.

This is a traditional exclusive lock that blocks all other read and write operations until the write operation is complete.

Stamped Approach:

Every lock acquisition returns a stamp, which is a long value representing the state of the lock.

This stamp is used to release the lock or to convert one type of lock into another (e.g., from an optimistic read to a write lock).

abstract static class Node CLH Nodes

```
volatile Node prev;       // initially attached via casTail
volatile Node next;       // visibly nonnull when signallable
Thread waiter;            // visibly nonnull when enqueued
volatile int status;      // written by owner, atomic bit ops by others
```

public long readLock()
public void unlockRead(long stamp)
public long writeLock()
public void unlockWrite(long stamp)

Usage

```
import java.util.concurrent.locks.StampedLock;

public class StampedLockExample {
    private final StampedLock lock = new StampedLock();
    private double x, y;

    // Writing method (exclusive access)
    public void move(double deltaX, double deltaY) {
        long stamp = lock.writeLock();
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            lock.unlockWrite(stamp);
        }
    }

    // Reading method (optimistic read)
    public double distanceFromOrigin() {
        long stamp = lock.tryOptimisticRead();
        double currentX = x, currentY = y;
        if (!lock.validate(stamp)) { // Check if a write occurred during the read
            stamp = lock.readLock();
            try {
                currentX = x;
                currentY = y;
            } finally {
                lock.unlockRead(stamp);
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY);
    }
}
```

function
Function

Function

package java.util.function;

public interface **Function**<T, R>　　函数式接口，接收 T，返回 R 结果

　　java.lang.Runnable

static <T> Function<T, T> identity()　　返回一个原样返回的匿名函数

default <V> Function<T, V> andThen(Function<? super R, ? extends V> after)　　调用当前函数，　将当前函数的返回值作为 after 函数的参数

default <V> Function<V, R> compose(Function<? super V, ? extends T> before)　调用 before 函数，将 before 的返回值作为当前函数的参数

R apply(T t);

IntFunction

package java.util.function;

public interface **IntFunction**<R>　　函数式接口，接收 int 参数，返回 R 结果

R apply(int value);

LongFunction

package java.util.function;

public interface **LongFunction**<R>　　函数式接口，接收 long 参数，返回 R 结果

R apply(long value);

ToDoubleFunction

package java.util.function;

public interface **ToDoubleFunction**<T>　　函数式接口，接收 T 参数，返回 Double 结果

ToIntFunction

package java.util.function;

public interface **ToIntFunction**<T>　　函数式接口，接收 T 参数，返回 Int 结果

ToLongFunction

package java.util.function;

public interface **ToLongFunction**<T>　　函数式接口，接收 T 参数，返回 Long 结果

LongToIntFunction

package java.util.function;

public interface **LongToIntFunction**　　函数式接口，接收 long 参数，返回 int 结果

LongToDoubleFunction

package java.util.function;

public interface **LongToDoubleFunction**　　函数式接口，接收 long 参数，返回 double 结果

IntToLongFunction

package java.util.function;

public interface **IntToLongFunction**　　函数式接口，接收 int 参数，返回 long 结果

IntToDoubleFunction

package java.util.function;

public interface **IntToDoubleFunction**　　函数式接口，接收 int 参数，返回 double 结果

BiFunction

BiFunction

package java.util.function;

public interface **BiFunction**<T, U, R>　　函数式接口，接收 T，U 两个参数，返回 R 结果

R apply(T t, U u);

default <V> BiFunction<T, U, V> andThen(Function<? super R, ? extends V> after)

ToDoubleBiFunction

package java.util.function;

public interface **ToDoubleBiFunction**<T, U>　函数式接口，接收 T，U 两个参数，返回 double 结果

ToIntBiFunction

package java.util.function;

public interface **ToIntBiFunction**<T, U>　　函数式接口，接收 T，U 两个参数，返回 Int 结果

ToLongBiFunction

package java.util.function;

public interface **ToLongBiFunction**<T, U>　　函数式接口，接收 T，U 两个参数，返回 long 结果

Consumer

Consumer

package java.util.function;

public interface **Consumer**<T>　　函数式接口，接受一个输入参数 T，并且无返回值，相当于消费者

void accept(T t);　　函数式接口的函数方法，传入一个任意类型，无返回值

default Consumer<T> andThen(Consumer<? super T> after)　　可以传入一个 Consumer，返回组合了两个 Consumer 后的 Consumer，传入的 Consumer 不能为 null

DoubleConsumer

package java.util.function;

public interface **DoubleConsumer**　函数式接口，接收 Double 参数，不返回结果

void accept(double value);

default DoubleConsumer andThen(DoubleConsumer after)

IntConsumer

package java.util.function;

public interface **IntConsumer**　　函数式接口，接收 Int 参数，不返回结果

void accept(int value);

default IntConsumer andThen(IntConsumer after)

LongConsumer

package java.util.function;

public interface **LongConsumer**　函数式接口，接收 long 参数，不返回结果

void accept(long value);

default LongConsumer andThen(LongConsumer after)

ObjLongConsumer

package java.util.function;

public interface **ObjLongConsumer**<T>　接收 T，long 参数，不返回结果

ObjIntConsumer

package java.util.function;

public interface **ObjIntConsumer**<T>　接收 T，int 参数，不返回结果

ObjDoubleConsumer

package java.util.function;
public interface **ObjDoubleConsumer**<T> 接收 T，double 参数，不返回结果

BiConsumer

BiConsumer

package java.util.function;
@FunctionalInterface
public interface **BiConsumer**<T, U> 函数式接口，接收 T，U 参数，不返回结果

Supplier

Supplier

package java.util.function;
public interface **Supplier**<T> 函数式接口，无参数，返回 T 结果
T get();

LongSupplier

package java.util.function;
public interface **LongSupplier** 函数式接口，无参数，返回 long 结果

Binary Operator

BinaryOperator

package java.util.function;
public interface **BinaryOperator**<T> extends BiFunction<T,T,T> 函数式接口， 接受 T，T 两个同类型参数，返回同类型 T 结果
public static <T> BinaryOperator<T> minBy(Comparator<? super T> comparator)
public static <T> BinaryOperator<T> maxBy(Comparator<? super T> comparator)

LongBinaryOperator

package java.util.function;
public interface **LongBinaryOperator** 接收 long，long 两个同类型参数，返回 long 结果

Unary Operator

UnaryOperator

package java.util.function;
public interface **UnaryOperator**<T> extends Function<T, T> 接收 T 参数，返回同类型 T 结果
static <T> UnaryOperator<T> identity()

LongUnaryOperator

package java.util.function;
public interface **LongUnaryOperator** 接收 long 参数，返回 long 结果

IntUnaryOperator

package java.util.function;
public interface **IntUnaryOperator** 接收 int 参数，返回 int 结果

Predicate

Predicate

package java.util.function;
public interface **Predicate**<T>　　函数式接口，接收参数 T，返回 boolean 结果

boolean test(T t);　　断言 是否成立
default Predicate<T> and(Predicate<? super T> other)　组合其他断言
default Predicate<T> negate()　　返回一个完全相反的断言
default Predicate<T> or(Predicate<? super T> other)　　或关系，组合一个断言
static <T> Predicate<T> isEqual(Object targetRef)　　使用 equals 判断参数是否相同
static <T> Predicate<T> not(Predicate<? super T> target)　　返回一个完全相反的断言

LongPredicate

package java.util.function;
public interface **LongPredicate**　　函数式接口，接收参数 long，返回 boolean 结果

logging

Logger

package java.util.logging;
public class **Logger**　　日志

Logger 的默认级别定义是在 jre 安装目录下
jre/lib/logging.properties >>
# 缺省的全局级别.
.level= INFO

# 控制台级别
java.util.logging.ConsoleHandler.level = INFO
java.util.logging.ConsoleHandler.formatter = java.util.logging.SimpleFormatter

public static Logger getLogger(String name)　　获取日志对象
public static Logger getLogger(String name, String resourceBundleName)　　获取日志对象
public void severe(String msg)　　打印日志
public void warning(String msg)　　打印日志
public void info(String msg)　　打印日志

Level

package java.util.logging;
public class **Level** implements java.io.Serializable　日志级别
public static final Level OFF = new Level("OFF",Integer.MAX_VALUE, defaultBundle);　　关闭日志记录
public static final Level SEVERE = new Level("SEVERE",1000, defaultBundle);　　最高值
public static final Level WARNING = new Level("WARNING", 900, defaultBundle);
public static final Level INFO = new Level("INFO", 800, defaultBundle);　　Logger 默认的级别是 INFO，比 INFO 更低的日志将不显示
public static final Level CONFIG = new Level("CONFIG", 700, defaultBundle);
public static final Level FINE = new Level("FINE", 500, defaultBundle);
public static final Level FINER = new Level("FINER", 400, defaultBundle);
public static final Level FINEST = new Level("FINEST", 300, defaultBundle);　　最低值
public static final Level ALL = new Level("ALL", Integer.MIN_VALUE, defaultBundle);　　启用所有消息的日志记录

prefs

```
package java.util.prefs;

/**
 * Static methods for translating Base64 encoded strings to byte arrays        用于将 Base64 编码字符串转换为字节数组的静态方法，反之亦然。
 * and vice-versa.
 *
 * @author  Josh Bloch
 * @see     Preferences
 * @since   1.4
 */
class Base64            base64 编解码类
```

regex

Matcher

```
package java.util.regex;
public final class Matcher implements MatchResult

public boolean find()       为 true 表示匹配到，如果匹配到多个，下一次调用 偏移到下一个匹配结果，每一次匹配可以理解为一组，从 0 开始，没有匹配到更多的返回 false
public boolean find(int start)  从指定位置开始 匹配，此种方式不会移动指针
public int start()          匹配字符串 第一个字符索引
public int end()            匹配字符串 最后一个字符索引
public String group()                   当前 regex 匹配到的字符串（必须先调用 find，和 find 使用同一个 matcher）
public String group(int group)          0 为整个字符串，其他表示当前 regex 匹配到第几组的字符串，正则括号的内容，没匹配到为 null（必须先调用 find，和 find 使用同一个 matcher）
                        如果出现分组(xxx)+  匹配多个的情况，此分组只会匹配第一种情况
public String replaceFirst(String replacement)    替换第一次匹配的数据
public String replaceAll(String replacement)      替换所有匹配的数据
public int regionStart()        匹配区域的开始，默认为 0
public int regionEnd()          匹配区域的结束，默认为字符串的 length
public boolean matches()  是否完全匹配
public Matcher reset()        充值指针
public static String quoteReplacement(String s)
    Matcher.quoteReplacement(String s)
        Escapes characters in a replacement string that have special meanings in replacement patterns.
        When using String.replaceAll() or Matcher.replaceAll(),
        It ensures that characters such as $ (which is used for group references) and \ (escape character) are treated as literal values in replacement operations.
    Pattern.quote(String s)
        Escapes all special regex characters in a given string so that it can be used as a literal pattern.
        It ensures that all regex metacharacters (. ^ $ * + ? { } [ ] \ | ( )) are treated as literal characters in the pattern.
    Escape String
        When printing a regex string, it displays as "\*".
        However, the actual regex in memory should be "\\*", since the backslash "\" is an escape character in Java strings, but the asterisk "*" is not.

        Original File: "\*"
```

Print result of read String: "*" (iteral)
Pass the read string as a regex parameter: "*" (iteral and valid)

Pattern

package java.util.regex;
public final class **Pattern** implements java.io.Serializable 正则

public static Pattern compile(String regex)
根据正则 返回 Pattern 实例（因为 Pattern 构造器是私有的）
public static boolean matches(String regex, CharSequence input)
是否完全匹配正则，matcher 的 find 是部分匹配
// Pattern.matches("\\d+","2223"); 2223 --> true 2223aa --> false 22bb23 --> false

public static String quote(String s)
Matcher.quoteReplacement(String s)
Escapes characters in a replacement string that have special meanings in replacement patterns.
When using String.replaceAll() or Matcher.replaceAll(),
It ensures that characters such as $ (which is used for group references) and \ (escape character) are treated as literal values in replacement operations.
Pattern.quote(String s)
Escapes all special regex characters in a given string so that it can be used as a literal pattern.
It ensures that all regex metacharacters (. ^ $ * + ? { } [ ] \ | ( )) are treated as literal characters in the pattern.
Escape String
When printing a regex string, it displays as "*".
However, the actual regex in memory should be "*", since the backslash "\" is an escape character in Java strings, but the asterisk "*" is not.

Original File: "*"
Print result of read String: "*" (iteral)
Pass the read string as a regex parameter: "*" (iteral and valid)

public Matcher matcher(CharSequence input) 匹配字符串

Usage

```
Pattern pattern = Pattern.compile("\\w+");
Matcher matcher = pattern.matcher("hello abc bbc cbc ccc");
//find 向前迭代
while(matcher.find()){
    System.out.println(matcher.group());
}
```

stream

Collectors

package java.util.stream;
public final class **Collectors**
public static <T> Collector<T, ?, List<T>> toList()
public static <T> Collector<T, ?, Set<T>> toSet()
public static <T, K, U>Collector<T, ?, Map<K,U>> toMap(

Function<? super T, ? extends K> keyMapper,
Function<? super T, ? extends U> valueMapper,
BinaryOperator<U> mergeFunction // Merge values for duplicated keys
)
The map.values() is immutable. the return map is modifiable.
Example:

```
list.stream().collect(
    Collectors.toMap(Contact::getContactId, Function.identity())
    );
Map<Integer, List<String>> collect = people.stream().collect(
    Collectors.toMap(
        Person::getAge,
        item -> new ArrayList<>(Arrays.asList(item.getName())),
        (v1, v2) -> {
                v1.addAll(v2);
                return v1;
        }
    )
);
List<person> list = Arrays.asList(p1, p2, p3);
Map<String, Integer> map = list.stream().collect(Collectors.toMap(person::getName, person::getAge));
```

public static <T, C extends Collection<T>> Collector<T, ?, C> toCollection(Supplier<C> collectionFactory)
Example:

```
List<String> linkedListResult = list.stream().collect(Collectors.toCollection(LinkedList::new))
```

public static<T,A,R,RR> Collector<T,A,RR> collectingAndThen(Collector<T,A,R> downstream, Function<R,RR> finisher)
First, pass all elements in the stream to the first parameter, and then pass the generated set to the second parameter for processing.
Example:

```
(List)implementations.stream().distinct().collect(Collectors.collectingAndThen(Collectors.toList(),
Collections::unmodifiableList));
```

public static <T> Collector<T, ?, Optional<T>> maxBy(Comparator<? super T> comparator)
Example:

```
List<Integer> num = Arrays.asList(1, 2, 3, 4, 5, 6);
Integer max = num.stream().collect(Collectors.maxBy((x,y)->x>y?1:-1)).get();
```

public static <T, K> Collector<T, ?, Map<K, List<T>>> groupingBy(Function<? super T, ? extends K> classifier)
Example:

```
List<person> list = Arrays.asList(p1, p2, p3);
Map<String, List<person>> collect1 = list.stream().collect(Collectors.groupingBy(p -> p.getName()));
```

public static <T> Collector<T, ?, Map<Boolean, List<T>>> partitioningBy(Predicate<? super T> predicate)
Example:

```
List<Integer> num = Arrays.asList(1, 2, 3, 4, 5, 6);
Map<Boolean, List<Integer>> map = num.stream().collect(Collectors.partitioningBy(x -> x % 2 == 0));
```

public static Collector<CharSequence, ?, String> joining(CharSequence delimiter)
Concatenate strings from a stream.
Example:

```
List<person> list = Arrays.asList(p1, p2, p3);
String str = list.stream().map(person::getName).collect(Collectors.joining(","));
```

usage
Collectors.groupingBy(Product::getCategory) 属性值分组 -- Object
//{ "啤酒":[{"category":"啤酒","id":4},{"category":"啤酒","id":5}], "零食":[{"category":"零食","id":1 },{"category":"零食

","id":2}] }
Collectors.groupingBy(item -> item.getCategory() + "_" + item.getName()) 拼接值分组 -- Object
//{ "零食_月饼":[{"category":"零食","id":3,"name":"月饼"}], "零食_面包":[{"category":"零食","id":1,"name":"面包" }] }
Collectors.groupingBy(item -> { 条件分组
//{"other":[{"id":3,"num":3},{"id":4, "num":3}, 3":[{"id":1, "num":1 },{"id":2,"num":2}]}
if(item.getNum() < 3) {
return "3";
}else {
return "other";
}
})
Collectors.collectingAndThen(Collectors.toCollection(()->new TreeSet<>(Comparator.comparing(WorkItem::getWorkItemId))) , ArrayList::new) 先进行结果集的收集，然后将收集到的结果集进行下一步的处理

Stream

package java.util.stream;
public interface **Stream<T>** extends BaseStream<T, Stream<T>>

public static<T> Stream<T> of(T... values)
Stream.of(arr) Convert original arr as a single element to an array.
Arrays.stream(arr) Convert original arr to a new complete array.
public static<T> Stream<T> empty() Return an empty stream
public static<T> Stream<T> iterate(final T seed, final UnaryOperator<T> f)
Params:
seed – the initial element
f – a function to be applied to the previous element to produce a new element
public static<T> Stream<T> generate(Supplier<? extends T> s)
Stream<T> distinct(); Using the equals method to duplicate
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
<R> Stream<R> flatMap(Function<? super T, ? extends Stream<? extends R>> mapper)
Example:
[[a,b,c],[d,e,f],[x,y,z]] => [a,b,c,d,e,f,x,y,z]
words.stream().map(w -> w.split("")).flatMap(Arrays::stream) .distinct().collect(Collectors.toList());
Stream<String[]> => Stream<String>

IntStream mapToInt(ToIntFunction<? super T> mapper);
// mapToInt(num -> Integer.parseInt(num))
Time Complexity: O(n)
LongStream mapToLong(ToLongFunction<? super T> mapper);
<U> Stream<U> mapToObj(IntFunction<? extends U> mapper);

Stream<T> sorted();
Sort the current stream
Stream<T> sorted(Comparator<? super T> comparator);

```
List<User> collect1 =
    users.stream().sorted(Comparator.comparing(User::getAge, Comparator.reverseOrder()))
    .collect(Collectors.toList());
```

```
List<User> collect =
    users.stream().sorted(Comparator.comparing(User::getAge).reversed())
    .collect(Collectors.toList());
List<User> collect2 =
    users.stream().sorted((x, y) -> y.getAge().compareTo(x.getAge()))
    .collect(Collectors.toList());
map.entrySet().stream()
    .sorted(Map.Entry.comparingByValue())
    .collect(Collectors.toMap(
        Map.Entry::getKey,
        Map.Entry::getValue,
         (e1, e2) -> e1, // Merge function in case of duplicate keys
        LinkedHashMap::new // Preserve the order by using a LinkedHashMap
    ));
```

Stream<T> distinct();

Stream<T> filter(Predicate<? super T> predicate);
void forEach(Consumer<? super T> action);
Optional<T> reduce(BinaryOperator<T> accumulator)
T reduce(T identity, BinaryOperator<T> accumulator);
Params:
identity – The return value of the last function execution, with the initial value being the value of the first element
accumulator – an associative, non-interfering, stateless function for combining two values
Example:
`reduce(999, (prev, next) -> prev += next)`

Optional<T> findAny();
FindAny does not focus on the order of elements and provides better performance in parallel stream operations because it can search for elements in multiple threads in parallel.
Optional<T> findFirst(); Search according to the natural order of the flow
boolean anyMatch(Predicate<? super T> predicate);
boolean allMatch(IntPredicate predicate);

<R, A> R collect(Collector<? super T, A, R> collector);
<A> A[] toArray(IntFunction<A[]> generator) Convert the stream to a specified array
Example:
Integer[] integers = integerStream.toArray(Integer[]::new);
Object[] toArray();
Time Complexity: O(n)
default List<T> toList()
The toList() method introduced in Java 16 returns an immutable list.
Stream<T> limit(long maxSize);

<R, A> R collect(Collector<? super T, A, R> collector);
Time Complexity: O(n)
<R> R collect(Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner);
Params:
supplier
Provides a new mutable container object that will be returned as the final result.
For a parallel execution, this function may be called multiple times and must return a fresh value each
accumulator

<container, element>
Processes each element in the stream and adds it to the result object.
combiner
Merges partial results from parallel stream execution.
<container, container>
Example:

```
List<String> strList = Stream.builder().add("aaa").add("bbb").add("ccc").build().parallel()
.collect(
    () -> new ArrayList<String>(),
    (list, str) -> list.add(String.valueOf(str)),
    (list1, list2) -> list1.addAll(list2)
    );
```

IntStream

package java.util.stream;
public interface **IntStream** extends BaseStream<Integer, IntStream>

public static IntStream of(int... values)
public static IntStream range(int startInclusive, int endExclusive)
Return a new IntStream: [startInclusive, endExclusive)
Example:

```
IntStream.range(0, str1.length).allMatch(i -> str1[i].equals(str2[i]));
```

public static IntStream rangeClosed(int startInclusive, int endInclusive)
Return a new IntStream: [startInclusive, endExclusive]

OptionalInt reduce(IntBinaryOperator op)
int reduce(int identity, IntBinaryOperator op)
int sum()
IntStream distinct();
IntStream sorted();
IntStream limit(long maxSize);
IntStream skip(long n);
OptionalInt min();
OptionalInt max();
long count();
OptionalDouble average();
boolean anyMatch(IntPredicate predicate);
boolean allMatch(IntPredicate predicate);
boolean noneMatch(IntPredicate predicate);
int[] toArray()

Stream<Integer> boxed()
All elements in the stream are boxed, which is exactly equal to mapToObj (Integer:: valueOf);
Time Complexity: O(n)
<U> Stream<U> mapToObj(IntFunction<? extends U> mapper)
<R> R collect(Supplier<R> supplier, ObjIntConsumer<R> accumulator, BiConsumer<R, R> combiner);
Unlike Stream::collect, it needs to be used with IntStream::boxed method

StreamSupport

package java.util.stream;

public final class **StreamSupport**　流支持

public static <T> Stream<T> stream(　创建 ReferencePipeline.Head 引用流抽象

Spliterator<T> spliterator,　分流器

boolean parallel　并行化

)

ReferencePipeline

package java.util.stream;

abstract class **ReferencePipeline**<P_IN, P_OUT>　pipeline 里对于中间操作和源头实现的抽象类

extends AbstractPipeline<P_IN, P_OUT, Stream<P_OUT>>

implements Stream<P_OUT>

static class Head<E_IN, E_OUT> extends ReferencePipeline<E_IN, E_OUT>　自继承接口，头部操作抽象

Head(Supplier<? extends Spliterator<?>> source, int sourceFlags, boolean parallel)

Head(Spliterator<?> source, int sourceFlags, boolean parallel)

public final <R> Stream<R> map(Function<? super P_OUT, ? extends R> mapper)　map 操作

StreamSpliterators

package java.util.stream;

class **StreamSpliterators**　流分流器工具

abstract static class InfiniteSupplyingSpliterator<T> implements Spliterator<T>　无穷供给分流器工具

public boolean tryAdvance(Consumer<? super T> action)　提供者作为参数，消费

zip

Deflater

package java.util.zip;

public class **Deflater**　同时使用了 LZ77 算法与哈夫曼编码的一个无损数据压缩算法

GZIPOutputStream

package java.util.zip;

public class **GZIPOutputStream** extends DeflaterOutputStream

The GZIPOutputStream class is used to compress data using the GZIP (GNU zip) file format.

Compression

```
import java.io.FileOutputStream;
import java.io.GZIPOutputStream;
import java.io.OutputStream;
import java.nio.charset.StandardCharsets;

public class GZIPExample {
    public static void main(String[] args) {
        String data = "This is some data to compress using GZIP.";
        try (OutputStream fos = new FileOutputStream("compressed-data.gz");
             GZIPOutputStream gzipOut = new GZIPOutputStream(fos)) {

            byte[] input = data.getBytes(StandardCharsets.UTF_8);
            gzipOut.write(input);

            // Automatically calls close on fos when gzipOut is closed.
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Decompression

```
import java.io.FileInputStream;
import java.io.GZIPInputStream;
import java.io.InputStream;
import java.nio.charset.StandardCharsets;

public class GZIPDecompressExample {
    public static void main(String[] args) {
        try (InputStream fis = new FileInputStream("compressed-data.gz");
             GZIPInputStream gzipIn = new GZIPInputStream(fis)) {

            // Read the decompressed data
            StringBuilder decompressedData = new StringBuilder();
            byte[] buffer = new byte[1024];
            int bytesRead;
            while ((bytesRead = gzipIn.read(buffer)) != -1) {
                decompressedData.append(new String(buffer, 0, bytesRead, StandardCharsets.UTF_8));
            }

            System.out.println("Decompressed Data: " + decompressedData.toString());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Inflater

package java.util.zip;

public class **Inflater**

This class provides support for general purpose decompression using the popular ZLIB compression library.

The ZLIB compression library was initially developed as part of the PNG graphics standard and is not protected by patents.

It is fully described in the specifications at the java.util.zip package description

InflaterInputStream

package java.util.zip;

public class **InflaterInputStream** extends FilterInputStream

InflaterInputStream is designed to read compressed data and decompress it using the DEFLATE compression algorithm.

It acts as a wrapper around an input stream, decompressing the data as it's read.

Compress a File

```
import java.io.FileOutputStream;
import java.io.FileInputStream;
import java.util.zip.DeflaterOutputStream;

public class CompressFile {
    public static void main(String[] args) throws Exception {
        String input = "Hello, this is compressed text!";
        try (FileOutputStream fos = new FileOutputStream("compressed.dat");
             DeflaterOutputStream dos = new DeflaterOutputStream(fos)) {
            dos.write(input.getBytes());
        }
    }
}
```

Decompress a File

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.util.zip.InflaterInputStream;

public class DecompressFile {
    public static void main(String[] args) throws Exception {
        try (FileInputStream fis = new FileInputStream("compressed.dat");
             InflaterInputStream iis = new InflaterInputStream(fis);
```

```
             FileOutputStream fos = new FileOutputStream("decompressed.txt")) {

            byte[] buffer = new byte[1024];
            int bytesRead;
            while ((bytesRead = iis.read(buffer)) != -1) {
                fos.write(buffer, 0, bytesRead);
            }
        }
    }
}
```

ZStreamRef

package java.util.zip;

class **ZStreamRef** A reference to the native zlib's z_stream structure.

ZipInputStream

package java.util.zip;

public class **ZipInputStream** extends InflaterInputStream implements ZipConstants

public byte[] readAllBytes() throws IOException