Java Tutorials

# Java

**Why Java**

Prior to Java the computer languages were **procedural** i.e., it was focused on functions to achieve solutions.

Procedural programming is a programming paradigm based on the concept of procedure(function) calls. In procedural languages, programs are organized as sequences of instructions that are executed step by step. These instructions are typically grouped into procedures (also called functions, subroutines, or methods) that can be called as needed.

Java on the other hand is **object-oriented** language i.e., its focused-on data and the change of state of data. In Java instead of just focussing on what needs to be done we also take care of which entity the change has to happen. For e.g., in Procedural programming, if we need to find area of a box, we will just write a function to do so. In Java we will create a class of Box with properties as its sides and then create function so that when any of the properties change the state of Box changes.

**How Java works?**

In Java the code we write is not directly executed. The .java files are converted to .class files(byte code) which is understood by JVM , which then runs the .class files and convert it into binary which is understood by OS/Processor.

**JVM** is responsible for converting **bytecode to machine-specific code**. It is also platform-dependent and performs many functions, including memory management and security. In addition, JVM can run programs that are written in other programming languages that have been converted to Java bytecode.

Bytecode is a low-level, platform-independent instruction set, acting as an intermediate language between source code and machine code, designed for efficient execution by a software interpreter or virtual machine. Bytecode is not directly executable by a CPU like machine code; instead, it's a set of instructions that a virtual machine (like the Java Virtual Machine or JVM) interprets and executes.

Java itself is platform independent, but the JVM which executes the code is Platform dependent/OS dependent. This is how Java gets the platform independence

**JRE** (Java Runtime Environment) is an installation package that provides an environment to **only run(not develop) the java program**(or application)onto your machine. JRE is only used by those who only want to run Java programs that are end-users of your system

**JDK** (Java Development Kit) is a Kit that provides the environment to **develop and execute(run) the Java program**. JDK is a kit(or package) that below items and more

A diagram of a computer program

AI-generated content may be incorrect.

|  |  |  |  |
| --- | --- | --- | --- |
| **Development Tools**   * **javac**: The Java compiler that converts .java files to bytecode (.class files) * **java**: The launcher for Java applications * **javadoc**: Tool for generating documentation from source code comments * **jar**: Tool for packaging Java files into JAR (Java Archive) files * **javap**: The Java class file disassembler * **jdb**: Java Debugger * **jconsole**: Monitoring and management console * **jshell**: Java REPL (Read-Eval-Print Loop) interactive shell | **Java Virtual Machine (JVM)**   * **Class Loader Subsystem** * **Runtime Data Areas**   + Heap Memory   + Stack Memory   + Method Area (non-Heap)   + PC Registers   + Native Method Stacks * **Execution Engine**   + Interpreter   + Just-In-Time (JIT) Compiler   + Garbage Collector * **Native Method Interface (JNI)** * **Native Method Libraries** | **Java Class Libraries**   * java.lang * java.io * java.util * java.net * java.math * java.awt * javax packages * etc. | **Java Standard Extensions**   * JavaFX * JDBC (Java Database Connectivity) * JNDI (Java Naming and Directory Interface) * etc. |

## JIT

For faster operation java utilizes native code. Native code is the code executed directly by processor.

Java code

    public int add(int *a*, int *b*) {

        return *a* + *b*;

    }

Native code

**x86 (Intel/AMD) Example: For a simple integer addition:**

mov eax, [ebp+8] ; Load first parameter into eax register

add eax, [ebp+12] ; Add second parameter to eax

**ARM Example:** For the same operation:

ldr r0, [r7, #8] ; Load first parameter into r0 register

ldr r1, [r7, #12] ; Load second parameter into r1

add r0, r0, r1 ; Add r1 to r0

Java 🡪(by javac)class files 🡪(by java) byte code 🡪 (by JIT)Native code for the OS

JIT is responsible for performance optimization of java-based applications during run time. At run time, the JVM loads the class files, determines the semantics of each individual bytecode, and performs the appropriate computation. The additional processor and memory usage during interpretation means that a Java application performs more slowly than a native application. The JIT compiler helps improve the performance of Java programs by compiling bytecodes into native machine code at run time.

The JIT compiler comes into play specifically for frequently executed code (called "hotspots"). So, say you have given a bytecode to your JVM which has thousands of lines. The JVM is not going to convert all your thousand lines of bytecode into the machine code immediately. It is only going to do the conversion on the fly based upon the method that you are trying to invoke

1. When your Java program first runs, the JVM interprets the bytecode instructions one by one. This interpretation process has overhead since the JVM needs to:
   * Read each instruction
   * Determine what it means
   * Execute the corresponding operations
2. During execution, the JVM's profiler monitors which parts of code are executed most frequently.
3. When it identifies a "hotspot" (methods or code blocks that are run repeatedly), the JIT compiler activates and compiles that specific portion of bytecode into highly optimized native machine code.
4. Subsequent calls to that code bypass the interpretation step entirely and execute the native code directly, which is much faster.

This approach offers several advantages:

* Only the performance-critical parts of the application get compiled, saving resources
* The JIT can apply runtime-specific optimizations based on actual execution patterns
* The program starts up quickly (no need to compile everything upfront)
* Performance improves the longer the application runs

This is why Java applications often have a **"warm-up"** period where performance gradually improves as the JIT identifies and optimizes the hotspots. For long-running applications like servers, this approach delivers near-native performance for the most critical code paths while maintaining Java's platform independence.

In simpler terms, JIT compilation involves translating Java bytecode into native machine code on-the-fly, just in time for execution. Unlike AOT (Ahead of Time) which compiles code before the program starts, offering fast loop execution but slower startup, JIT translates code on-the-fly as the program runs, providing faster startups and adapting to active code paths. This dynamic approach generates optimized native code for the actual execution context. This is why it is said that java has a **cold start** problem.

## Classes Objects

Java depends on classes to run program. Anything and everything in java must be defined in a class. For e.g., if we need to calculate area of box. We need to define a class of Box, create an object, and then calculate Area.

Class is a representation of real-life entity in Java.

public class Test {

    public static void main(String[] *args*) {

        Box b1 = new Box();

        b1.height = 6;

        b1.length = 5;

        b1.width = 4;

        System.out.println(b1.area());

    }

}

class Box {

    int height, width, length;

    int area() {

        return this.height \* this.length \* this.width;

    }

}

A class defines a new type of data. When you create a class, you are creating a new data type. You can use this type to declare objects or instance of that type. For e.g. Person can be a class and Amit an instance or object of Person.

A class is a template for an object, and an object is an instance of a class.

Variables defined within a class are called instance variables because each instance of the class (that is, each object of the class) contains its own copy of these variables.

## JAVA CLI

How to run JAVA program?

Create java Class file Amit.java

public class Test {

    public static void main(String[] *args*) {

        System.out.println("Hello World");

    }

}

Compile .java to .class by running 🡪 javac .\Test.java

Run the class🡪 java Test

In java11 we can run directly without compiling like *“java Amit.java*”. This is called single-file source-code.

We can also pass arguments to java program. For e.g. for below Test.java we can run as *java Test.java 123 ABC*

public class Test {

    public static void main(String[] *args*) {

        if (*args*.length > 0) {

            Stream.of(*args*).forEach(*s* -> {

                System.out.println("Your Args" + *s*);

            });

        } else {

            System.out.println("No Args passed");

        }

    }

}

## New and Java memory allocation

* In Java objects are stored in a memory location called Heap.
* When objects are declared just a reference variable is created pointing to nothing.
* Objects are created with new keyword. new operator dynamically allocates during runtime memory for an object and returns a reference to it.
* This reference is, essentially, the address in memory of the object allocated by new. This reference is then stored in the variable.
* So before new we can’t access its members (variables or methods)

class Box {

    int height, width, length;

    int area() {

        return this.height \* this.length \* this.width;

    }

}

        Box myBox;          *// declare object or create reference*

        myBox = new Box();  *// instantiate object or allocate a Box object*

A diagram of a box

AI-generated content may be incorrect. 

Whenever we use = operator with objects it does memory assignment.

* b1 = b2 means both points to same memory location. So if I change the state or property of one object, other will also get changed.
* If we do b1 = null b2 will still points to previous memory location
* If we do b1 = b2 and after that if we do b2 = b3, it will mean b1 still points to old location and will not change
* If b1 = b2 and b1.property is changed then b2.property will also change

Example below

public class Test {

    public static void main(String[] *args*) {

        Box b1 = new Box(11, 12, 13);

        Box b2 = b1; *// Both b1 and b2 points to same memory location*

        b1.height = 24; *// When one property of b1 is changed, b2 also gets changed*

        System.out.println(b2.height);

        b1 = null; *// But when b1 is pointed to null b2 still points to previous object*

        System.out.println(b2.height);

    }

}

class Box {

    int height, width, length;

    public Box(int *height*, int *width*, int *length*) {

        this.height = *height*;

        this.width = *width*;

        this.length = *length*;

    }

}

Once the object is created with new keyword(not needed for String) then only we can act on it and access its members

        Box box;

        box.toString(); *// This is not compile*

        Box box = null; *// With this we can trick Java into compiling*

        box.toString(); *// But will throw NullPointerException*

Primitives like int height; don’t need new operator as they are stored in stack instead of heap where objects are stored for better efficiency. new allocates memory for an object during run time. The advantage of this approach is that your program can create as many or as few objects as it needs during the execution of your program. However, since memory is finite, it is possible that new will not be able to allocate memory for an object because insufficient memory exists. If this happens, a run-time exception will occur.

**Java Memory management**

The JVM divides the memory into two parts: [stack](https://www.javatpoint.com/java-stack) memory and [heap](https://www.javatpoint.com/java-heap) memory. Stack is used to store the order of method execution and local variables while the heap memory stores the objects, and it uses dynamic memory allocation and deallocation.

1. **Heap** Objects
2. **Stack** This is the area of memory where local variables and method calls are going to be stored. So whenever a new method is being called, a new stack frame is going to be created on the stack, which will contain all the methods, local variables and parameters. And whenever the method returns the execution or completes the execution, the stack frame is going to be removed along with the local variables and the parameters.
3. **Method Area** This is the place where we are going to store all the class definitions, method definitions and other runtime constants like static variables are going to be stored.
4. **Native Heap** This is the area of memory where native libraries, and the code are going to be loaded are stored. JIT compiler is going to identify the hotspots inside my classes and methods, which are going to be executed many times or very frequently all such hotspots bytecode is going to be converted into a native executable code by the JIT compiler, and it is going to cache it inside a memory. And that memory is native heap.

When object are not used then it is garbage collected from Heap

<https://www.javatpoint.com/stack-vs-heap-java>

# Data Types Operators

## Primitive Types

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | boolean | 1bit |  |  |  |
| 2 | byte | 8 bits  1 byte | 2^8 | - 128 to 127 |  |
| 3 | char | 16 bits  2 bytes | 2^16 | Stores value as int 65,536 | So large to accommodate all special character, uses UNICODE |
| 4 | short | 16 bits  2 bytes | 2^16 | -32,768 to 32,767 |  |
| 4 | int | 32 bits  4 bytes | 2^32 | -2,147,483,648 to 2,147,483,647 |  |
| 6 | long | 64 bits 8 bytes | 2^64 |  |  |
| 7 | float | 32 bits  4 bytes | 2^32 |  |  |
| 8 | double | 64 bits 8 bytes | 2^64 |  |  |

Unlike objects **primitives have default values** inside a class and won’t give null pointer exception when used at class level. Default value of Boolean is false, char is blank space, and all numerical type is 0.0 or 0. However the default value is assigned when the primitives are member of a class. Not when they are local variables.

public class Test {

    static boolean bool;

    static int i;

    static char b;

    static float f;

    public static void main(String[] *args*) {

        System.out.println(String.valueOf(bool)); *// false*

        System.out.println(String.valueOf(i)); *// 0*

        System.out.println(String.valueOf("<>" + b + "<>")); *// <><>*

        System.out.println(String.valueOf(f)); *// 0.0*

    }

}

If we use uninitialized local variable, it will throw error.

public class Test {

    public static void main(String[] *args*) {

        boolean bool;

        int i;

        char b;

        float f;

*//Will not compile*

        System.out.println(String.valueOf(bool));

        System.out.println(String.valueOf(i));

        System.out.println(String.valueOf("<>" + b + "<>"));

        System.out.println(String.valueOf(f));

    }

}

Char uses Unicode, so we can represent char with \u to represent Unicode

public class Test {

    public static void main(String[] *args*) {

        char i = '\u5632';

        System.out.println(String.valueOf(i));  *// will print ?*

    }

}

Java defaults to integer in an expression i.e. When 2 bytes or short are added in an expression JVM automatically assigns an int for it to avoid any type mismatch

        byte i =4 ;

        byte j = 6;

        byte k = i+j; *// won't compile as it is treated as integer instead of byte*

But it doesn’t happen inside an expression as it is calculated compile time.

        byte b = 3 + 4;

        System.out.println(b);

But if we are out of range it will fail during compilation

        byte b = 139+4;  *//won't compile*

Octal number 0-7 starts with 0 int i = 027; used in embedded systems

Hexadecimal number int i = 0x1B3 ; 0-9 A-F

## Casting

When we assign a value of one primitive data type to another type, we call it as **type casting**.

In Java, there are two types of casting:

**Widening Casting (automatically/implicit)** - converting a smaller type to a larger type size. It is done automatically & safe because there is no chance to lose data.

byte -> short -> char -> int -> long -> float -> double

**Narrowing Casting (manually/explicit)** - converting a larger type to a smaller size type. It should be done manually by the programmer. If we do not perform casting then the compiler reports a compile-time error.

s

double -> float -> long -> int -> char -> short -> byte

The process to assigning one data type to another.

It’s of 2 types

1. Implicit casting -> Smaller type to larger. Nothing needed JVM takes care of it

        byte i =4 ;

        int j = i;

since char stores values as int

        char ch = 'A';

        int i = ch;

        System.out.println(i); *// will return 65*

1. Explicit casting Larger to smaller. In this case explicitly cast has to be mentioned

        int i =4 ;

        byte j = (byte) i;

When the cast is outside the range it goes on negative

        int i = 132;

        byte b = (byte) i;

        System.out.println(b); *// -124*

        long l = 2147483648l;

        int i = (int) l;

        System.out.println(i); *//-2147483648*

And if it is still out of the range the process continues in circle till it reaches end

        int i = 12345;

        byte b = (byte) i;

        System.out.println(b); *// 57*

## Wrapper Classes

Primitive to Object Boxing

Object to primitive Unboxing

        int i =3;

        Integer in = Integer.valueOf(i);  *//Boxing*

        i = in.intValue();  *//Unboxing*

String conversion

        byte x = 100;

        String s = Byte.toString(x);

        Byte byte1 = Byte.parseByte(s);

        System.out.println(x+ s+ byte1);

        String s = "2000";

        Integer i = Integer.parseInt(s);

java.lang.NumberFormatException: For input string: "a"

        String s = "a";

        Integer i = Integer.parseInt(s);

        System.out.println(i);

Wrapper classes do caching for small values to save memory.

ByteCache (-128 to 127)

ShortCache (-128 to 127)

LongCache (-128 to 127)

CharacterCache (0 to 127)

Which means when the value is within this value java will resue the values

        Integer i1 = 127;

        Integer i2 = 127;

        System.out.println(i1 == i2);   *//true*

        i1 = 128;

        i2 = 128;

        System.out.println(i1 == i2);   *//false*

## Operators and control Flow

**Pre increment/decrement operator** (++ --)value change first then assignment

        int i = 4;

        int j = ++i;  *//both will have 5*

        System.out.println("i " + i + " j " + j);

**Post increment/decrement operator** first assignment then value change

        int i = 4;

        int j = i--;  *//i will 3 while j will be 4*

        System.out.println("i " + i + " j " + j);

Concatenation Operator(+)

Only operator which is overloaded for String it concats for int it adds

        int i = 3;

        int j = 4;

        String s = "Hello";

        System.out.println(i + j);  *//7*

        System.out.println(i + j + s);  *// 7hello*

        System.out.println(s + i + j); *//Hello34*

Arithmetic operation + - \* / %

**Relational Operator** <, <=, >, >=, ==, !=

Can’t be used with boolean

System.out.println(true> false); *// won't compile*

Bitwise operator

|  |  |  |
| --- | --- | --- |
| & | Both true | Works for integer and boolean |
| && | Both true but if 1st is false won’t evaluate 2nd | Works only for boolean, better performance |
| | | Any one true | Works for integer and boolean |
| || | if 1st is true won’t evaluate 2nd | Works only for boolean, better performance |
| ^ | Both should be different one true one false |  |
| ~ | Converts to bit and Flip all bits | Works only on int |
| ! |  |  |

| and & works on int also it converts the int to binary and work on each of the digit

System.out.println(4 | 5);  *//5*

System.out.println(4 & 5);  *//4*

4 & 5 100

101

100 = 4

4 |5 100

101

101 =5

Assignment operator =, =+, =-

        int x, y, z;

        x = y = z = 22;

        int i = 3;

        System.out.println(i += 40); *//43*

        System.out.println(i += 6); *//49*

        System.out.println(i \*= 4); *//196*

        System.out.println(i /= 4); *//49*

        System.out.println(i %= 2); *//1*

**Ternary operator** Expression ? vaue1 : value2

Control Flow

If

else

else if

do

while

break

continue

return

## String

Strings are immutable, i.e. their values can’t be changed. When we assign new values to a string it just points to a new location in memory (thread pool), the original string stays in the pool.

        String s1 = "Hello";

        String s2 = "Hello";

        System.out.println(s1 == s2);   *//true*

        s1 = new String("hello");

        s2 = new String("hello");

        System.out.println(s1 == s2);   *//false*

Trick Here in line 2 sicne we are not assigning values s1 value is unchanged

        String s1 = "Hello";

        s1.concat(" World");

        System.out.println(s1); *// Prints “Hello”*

When string is constructed without using a constructor then string is stored in a special area of heap called **string pool** whose purpose is to maintain a set of unique strings this is called **interning**. When two strings have the same values then they point to the same location in the string pool hence the == method on two strings will return true. But this will not be the case when we use a specific constructor to create a string as shown below.

|  |  |
| --- | --- |
| String s1 = new String("Hello").intern();  String s2 = new String("Hello").intern();  System.out.println(s1 == s2); //True | String s1 = new String();  String s2 = new String();  s1 = "hello"; s2 = "hello";  System.out.println(s1 == s2); //True |

String can also be created from an array of bytes, char and integer. While passing array we can pass

**offset→** no of items that will be skipped from array

**count→** no elements taken from array

offset + count should be equal or less than the length of the array. Above params are optional for byte[] and char[] but mandatory for int[]. Let's say we create array of alphabets a -z and add them to string

        char[] ch = new char[26];

        byte[] by = new byte[26];

        int[] in = new int[26];

        int i = 0;

        for (char c = 'a'; c <= 'z'; c++, i++) {

            ch[i] = c;

            by[i] = (byte) c;

            in[i] = c;

        }

        String s1 = new String(ch, 0, 26); *// abcdefghijklmnopqrstuvwxyz*

        String s2 = new String(by, 2, 22);*// cdefghijklmnopqrstuvwx ab from beginning and yz from end skipped*

        String s3 = new String(in, 1, 25); *// count and offset is mandatory*

        System.out.println(s1);

        System.out.println(s2);

        System.out.println(s3);

**String concat**  String uses concat() or + operator to join two strings

1. Although we can do string + 3 i.e. concat int or other types with string using “+” but **str.concat(3** will not compile.
2. Similarly **str + null** will compile and give something like “Hello null” but **str.concat(null)** will give a null pointer exception. However in either case we can’t use uninitialized variables like **String str2; str + str2 ;** This will not compile**.**
3. “+” operator converts int or any other literal to String using to string but we can override using ()

        int i = 20, j = 20;

        System.out.println(i + j + " = " + i + j); *// 40 = 2020*

        System.out.println(i + j + " = " + (i + j)); *// 40 = 40*

Like Char String also uses Unicode

        String m = "\u004D";

        System.out.println(m);

        String unicodeName = "Hello Amit \u0021";

        System.out.println(unicodeName);

String methods

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Methods | Purpose | Supported by StringBuilder |  |
|  | intern() | Will force the string to use String pool |  | String hello = "Hello";          String obj = **new** String("Hello").intern();          System.out.println(hello==obj); |
|  | length() |  | y |  |
|  | valueOf | int intValue = 42;  String.valueOf(intValue); |  | Convert primitives to String |
|  | charAt() |  | y |  |
| Compare | equals() |  | Y |  |
| equalsIgnoreCase() |  | N |  |
| compareTo | "java".compareTo("java");  Result 0  "java".compareTo("python")  Result -6 |  | Compares the Unicode char and returns the difference. For difference between j and p is 6 with j smaller so returns -6. Can be used for sorting |
| isEmpty |  |  | Check for “” |
| isBlank |  |  | Check for “ “ |
| Search operations | startsWith() |  | N |  |
| endsWith() |  | N |  |
| substring() | sb.substring(0, 4)  Hell | y | Returns the string in the index provided first one inclusive , 2nd exclusive |
| indexOf() | sb= "Hello";  sb.indexOf("ll")  0 true -1 False | y | Returns the position where the string is found if not found returns -1 |
| contains() | sb.contains("s"); | N | One format takes String other char sequence |
| matches() | str.matches(".\*World.\*"); |  | Uses regex |
| Modification | replace() | "abc".replace('a', 'A');  "abc".replace("a", "A"); | N | Can use regex also like below is used to replace vowels  String originalString2 = "Java is fun!";  String replacedString4 = originalString2.replaceAll("a|e|i|o|u", "\*"); |
| toLowerCase() |  | N |  |
| toUpperCase() |  | N |  |
| trim() |  |  |  |
| strip() |  |  |  |
| stripLeading() |  |  |  |
| stripTrailing() |  |  |  |

Different ways of String empty check

**boolean** isEmpty = myString.isEmpty(); // can give null pointer if string is null

**boolean** isLengthZero = myString.length()==0; // can give null pointer if string is null

**boolean** isEqualEmpty = "".equals(myString); // no issue with null String

For blank check

String blankString = "       ";

**boolean** isBlankEmpty = blankString.isEmpty();

Format method is used to format the String by replacing the placeholder

        String message = "Hello, %s ! You have %d messages."; *// %s for replacing String , %d for int*

        String msgForAmit = String.format(message, "Amit", 3);

        String msgForShamit = String.format(message, "Shamit", 6);

        System.out.println(msgForAmit);

        System.out.println(msgForShamit);

        String message1 = "The price is $%.2f"; *// %.f to insert double or float 2 represent the no of decimal needed*

        String price1 = String.format(message1, 19.99);

        System.out.println(price1);

        String price2 = String.format(message1, 9.99);

        System.out.println(price2);

        String message2 = "The number is %5d"; *// %5d 5 for 5 spaces and d for int*

        String number1 = String.format(message2, 5);

        String number2 = String.format(message2, 15);

        System.out.println(number1);

        System.out.println(number2);

        String message3 = "My name is %3$s, I am %2$d years old, I live in %1$s.";  *//%2$ reresent 2nd argument will be inserted here*

        String result = String.format(message3, "NewYork", 25, "Nitin");

        System.out.println(result);

**Text Block**  This will print like with new line tab etc. and we don’t need to format it. It uses 3 quotes “”” “””

        String textBlock = """

                    <html>

                        <body>

                            <p> Hello World. </p>

                        </body>

                   </html>

                """;

        System.out.println(textBlock);

**String vs String buffer vs Builder**

String is a final class. All the fields as final except “private int hash”. This field contains the hashCode() function value. The hashcode value is calculated only when the hashCode() method is called for the first time and then cached in this field. Furthermore, the hash is generated using the final fields of String class with some calculations. So every time the hashCode() method is called, it will result in the same output. For the caller, it seems like calculations are happening every time but internally it’s cached in the hash field.

|  |  |  |
| --- | --- | --- |
| String | String buffer | StringBuilder |
| Immutable | Mutable | Mutable |
| Slow for concat operations | Fast for concat operations | Fastest for concat |
| String constant pool. | Uses heap |  |
| Thread safe | synchronized i.e. thread safe | non-synchronized i.e. not thread safe |

        int count = 10\_000;

        String st = "Hello";

        StringBuilder sb = new StringBuilder("Hello");

        StringBuffer sf = new StringBuffer("Hello");

        Long l1 = System.currentTimeMillis();

        for (int i = 0; i < count; i++) {

            st = st + i;

        }

        Long l2 = System.currentTimeMillis();

        System.out.println("Time taken " + (l2 - l1));

        l1 = System.currentTimeMillis();

        for (int i = 0; i < count; i++) {

            sb = sb.append(i);

        }

        l2 = System.currentTimeMillis();

        System.out.println("Time taken " + (l2 - l1));

        l1 = System.currentTimeMillis();

        for (int i = 0; i < count; i++) {

            sf = sf.append(i);

        }

        l2 = System.currentTimeMillis();

        System.out.println("Time taken " + (l2 - l1));

# Packages

Packages are an efficient way to organize Java classes. They are like directories and sub directories under which Java classes are organized systematically. Otherwise, there can be a naming conflict between classes with same name and it will really difficult to find our required classes.

package saha.amit;

public class Amit {

    public static void main(String[] args) {

        System.out.println("Hello Amit");

    }

}

Difference between package and directory is that package name must be added in the class and gives the context from where class can be loaded. For example we can still run above class if we remove package declaration by running command java saha/amit/Amit.java , which is like going to directory and running Java command. But with package declaration we can run the class from top directory with command java saha.amit.Amit. Notice . used instead of / and no .class

Note if we want to use

Package is also helpful when we import one class in one package to another package.

Suppose we have below folder structure for java files Amit.java and Shamit.java

src→saha →amit →Amit.java

→shamit →Shamit.java

package saha.amit;

public class Amit {

    public static void main(String[] args) {

        System.out.println("Hello Amit");

    }

}

package saha.shamit;

import saha.amit.Amit;

public class Shamit {

    Amit am = new Amit();

    public static void main(String[] args) {

        System.out.println("Hello Shamit");

    }

}

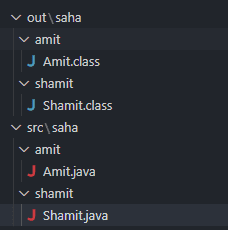
Notice since root package is saha, hence we know the context from where other classes can be loaded.

In java11 we can run directly without compiling like **java saha/amit/Amit.java**. This is called single-file source-code. But this will not work for Shamit.java i.e. **java saha/shamit/Shamit.java** which has reference to class Amit. So we have to compile first Amit class then Shamit class with command

javac saha/amit/Amit.java saha/shamit/Shamit.java and then run java saha.shamit.Shamit from src folder

Note Amit.java which doesn’t have any dependency on other file can be run as both **java saha/amit/Amit.java or java saha.amit.Amit**. However, for later javac has to be run which runs on .class file which is created after javac

By default the class files will be created next to .java files. If we want to keep our classes in a folder named “out” we have to use the -d option like **javac -d out saha\amit\Amit.java saha\shamit\Shamit.java**. After -d we mention the path where we want to create the class files. In our case it will give the below structure.



We may not always be in the folder from where we are running our command. This is not a problem for Amit.java which is single file source code and can be run from anywhere using java command.

java JavaModule/src/saha/amit/Amit.java

But classes like Shamit which have to run from class files need to be on classpath which is the root from where the package is starting. So for Shamit we need to be inside the “out” folder and run

java saha.shamit.Shamit

If we are ruining from location outside out we need to mention classpath or -cp option.

java -cp out saha.shamit.Shamit

if we go outside my current folder named “JAVA”. Then the command will be

java -cp JAVA/out saha.shamit.Shamit

We can also import an entire package using \* like below

import java.util.\*;

But this is not advisable as it will import all Classes in the package. Some say it will slow the compilation process however that is not true. But what it does is that since unwanted classes are imported it can create a conflict as two Classes For e.g. In below class if we import entire java.util and java.sql then compiler won’t know which class to load. Plus it makes the code cluttered.

import java.sql.\*;

import java.util.\*;

public class Amit {

    public static void main(String[] args) {

        Date date = new Date(0);  *//not compiling*

    }

}

If you want to use 2 classes with same name you have to use FQCN fully qualified class name for 2nd class

import java.util.Date;

public class Amit {

    public static void main(String[] args) {

        Date date = new Date();

        java.sql.Date date2 = new java.sql.Date(0);

    }

}

java.lang.\* is by default included in all Java classes, hence we don’t need to import any class from java.lang like System.out.println

Static variables in java.lang has to qualified with class name for e.g.

    public static void main(String[] args) {

        System.out.println();

        Integer.parseInt("6");

    }

Instead we can do Static import in that case we don’t gave to give class name before method call

import static java.lang.Integer.\*;

import static java.lang.System.\*;

public class Amit {

    public static void main(String[] args) {

        out.println();

        parseInt("6");

    }

Only when two methods have same name we have to use Class.method name for 2nd implementation.

In order pass arguments while running Java program, open run configuration in eclipse.



Add arguments separated with space.



From CLI just pas the arguments after file name java Test.java 123 ABC

    public static void main(String[] args) {

        if (args.length >0) {

            Stream.of(args).forEach(s->{

                System.out.println("Your Args"+s);

            });

        } else {

            System.out.println("No Args passed");

        }

    }

# Classes Object

## Members of a class

### Variables

Multiple variables can be declared like this also.

int i, j;

But during initialization only the first one is initialized. So in below code i is not initialized and if try to do any operation with i it will not compile and give error

        int i, j =3;

        System.out.println(i + j);  // i not initialized and will give error

To initialize both have to do like this

        int i=3, j =3;

or

        int x, y, z;

        x = y = z = 22;

If you assign value while declaring variable then all copies of the object will have same values, otherwise it will have default value.

public class Test {

    int i = 3;

    int j;

    public static void main(String[] args) {

        Test test1 = new Test();

        Test test2 = new Test();

        System.out.println("Test object i value --> "+ test1.i + " "+ test2.i);

        //Both object have same value for i =3 whihc was declared

        System.out.println("Test 2 -->"+ test1.j + " "+ test2.j);

        //Both object have same value value for j =0 whihc is default

    }

}

### Methods

Has params and return type.

### Constructors

A constructor tells how the object of a class will be instantiated and with what values. It can be tedious to initialize all the variables in a class each time an instance is created. So, we can assign the values during object creation.

    Box(int h, int b, int l) {

        this.height = h;

        this.breath = b;

        this.length = l;

    }

Box b1 = new Box(4,5,6);

Constructors look a little strange because they have no return type, not even void. This is because the implicit return type of a class’s constructor is the class type itself.

        Box myBox = new Box();

When we try above no constructor is needed. When you do not explicitly define a constructor for a class, then Java creates a default constructor for the class. Which is no argument constructor.

But if a contractor is already declared then no argument constructor will not be generated by default. So below will not compile as we have defined a constructor with arguments and no default constructor

public class Test {

    public static void main(String[] args) {

        Box myBox = new Box();          *// will not compile*

    }

}

class Box {

    int height,breath, length;

    Box(int height, int breath, int length) {

        this.height = height;

        this.breath = breath;

        this.length = length;

    }

}

In this case will have to explicitly mention no arguments constructors.

class Box {

    int height,breath, length;

    Box(){}

    Box(int height, int breath, int length) {

        this.height = height;

        this.breath = breath;

        this.length = length;

    }

}

### Instance Initialization/Code Blocks

Code blocks are piece of code which are executed when the class is instantiated. You can also create objects there.

Needed for

1. Anonymous class which don’t have constructors to handle initialization logic.
2. Also if you have any common logic for multiple constructor you can keep in code blocks

public class Test {

    {

        System.out.println("this is a code block");

        Test test = new Test();

    }

    public static void main(String[] args) {

    }

}

Code blocks can also be static. Static blocks are loaded when the Class is loaded in JVM for the 1st time. When you initialize class again with new it will not execute again. non-static code block will execute even time the class initialized with new

### Order Of Execution

When any class is instantiated

1. First any code block will execute e.g. “This is Box class code”.
2. Static block is executed first then non static block. (Static block will execute when the class is loaded in JVM for 1st time, non-static code block will execute even time the class initialized with new )
3. In case of multiple non static blocks, they are executed in the order they appear.
4. Then constructor code will execute e.g. “This is box-no argument constructor”.
5. Then any method is executed e.g., volume ().
6. In case of static method call code block and constructor is not called.

class Box {

    int height, breath, length;

    Box() { System.out.println("This is box no argumnt constructor");}

    {

  System.out.println("This is Box class code");  *//execute before constructor*

    }

    public int volume() {

        System.out.println("Calculating area");

        return this.height \* this.breath \* this.length;

    }

    public static void test() {

        System.out.println("Static method");

    }

}

## Static

Static variables stored in method area. Whenever we try to create the static variables inside the Java classes, behind the scenes, all the static variables are going to be stored inside a memory called method area. This is an area inside the JVM memory which is going to be shared by all the threads. When we try to execute a Java program, JVM first loads the Java class into the memory, and it will look for any static variables that are present inside the class. If they are present, then they are going to initialize and store them inside the memory called method area class area and the static variables that we store inside these memories are going to be remain and available until the program terminates.

All the objects that we create inside the Java are going to be stored inside the heap memory. Whereas the static variables that we are going to create will be stored inside the method area or class area. So, if you try to create too many static variables inside your classes, it may cause out of memory error.



Static Block

public class Test {

    static {

        System.out.println("This is static block"); *//Will execute before Main or anything.*

    }

    public static void main(String[] args) {

        System.out.println("Main");

    }

}

public class Test {

    static {

        System.out.println("This is static block");

        Test test = new Test();

        System.out.println(test);

        //Can create object but can't access non static method or variables

    }

}

Static block vs Non static block

public class Test {

    static {

        System.out.println("This is static block");

        //Will be called once when class is loaded

    }

    {

        System.out.println("Non static block");

        //Will be called every time object is created

    }

    public static void main(String[] args) {

        new Test();

        new Test();

    }

}

public class Test {

    public static void main(String[] args) {

        new Another();

        new Another();

    }

}

class Another {

    static {

        System.out.println("This is static block");

        //Will be called once when class is loaded

    }

    {

        System.out.println("Non static block");

        //Will be called every time object is created

    }

}

Static methods

public class Test {

    int i;

    static {

        method1();

        //Can access static method also from static blocks

    }

    static void method1(){

        System.out.println("This is a static method");

        //can't access i or any non static variables or methods like i or method2

    }

    void method2() {}

    public static void main(String[] args) {

        method1();

        Test.method1(); *// access in both ways*

    }

}

Static reference and creating objects in static block

public class Test {

    static Test test;

    static {

        System.out.println(Test.test); *//Here the object will be null*

        Test.test = new Test();

    }

    public static void main(String[] args) {

        System.out.println(Test.test);

        //Here since it is already initiated in static block will have a valid reference

    }

}

Infinite loop

public class Test {

    static Test test;

    {

        System.out.println(Test.test);

        Test.test = new Test();

    }

    public static void main(String[] args) {

        new Test();

        System.out.println(Test.test);

        //This will go in infinite recursive loop

    }

}

## This keyword

Sometimes a method will need to refer to the object that invoked it. To allow this, Java defines the this keyword. this can be used inside any method to refer to the current object

    Box(int height, int breath, int length) {

        this.height = height;

        this.breath = breath;

        this.length = length;

    }

This key word can only be used in non static context like non static block or method not in static block or method

public class Test {

    int i;

    {

        this.i =3; *//will compile*

    }

    static {

        this.i =3; *//won't compile*

    }

    public static void main(String[] args) {

        this.i =3; *//won't compile*

    }

}

## Overloading Methods

it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different. When this is the case, the methods are said to be overloaded, and the process is referred to as method overloading. When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to call.

class Box {

    int height, breath, length;

    public int volume(int height, int breath, int length) {

        return height \* breath \* length;

    }

    public int volume(int height, int breath) {

        return height \* breath \* this.length;

    }

}

Thus, overloaded methods must differ in the type and/or number of their parameters. While overloaded methods may have different return types, the return type alone is insufficient to distinguish two versions of a method. When Java encounters a call to an overloaded method, it simply executes the version of the method whose parameters match the arguments used in the call.

So below will not work.

class Box {

    int height, breath, length;

    public int volume(int height, int breath, int length) {

        return height \* breath \* length;

    }

    public String volume(int height, int breath, int length) {

        return (height \* breath \* length) +"Area";

    }

}

When an overloaded method is called, Java looks for a match between the arguments used to call the method and the method’s parameters. However, this match need not always be exact. In some cases, Java’s automatic type conversions can play a role in overload resolution.

public class Test {

    void test (){}

    void test (double d){}

    public static void main(String[] args) {

        int i = 5;

        Test tst = new Test();

        tst.test(i);

    }

}

Method overloading supports polymorphism because it is one way that Java implements the “one interface, multiple methods” paradigm. The value of overloading is that it allows related methods to be accessed by use of a common name. Thus, the function name represents the general action that is being performed. It is left to the compiler to choose the right specific version for a particular circumstance. You, the programmer, need only remember the general operation being performed.

## Call by Value/Reference

In general, there are two ways that a computer language can pass an argument to a subroutine.

1. **Call-by-value**. This approach copies the value of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.
2. **Call-by-reference**. In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

As you will see, Java uses call-by-value to pass all arguments. As we can see below that weather we pass local value, static value or instance value is not changing

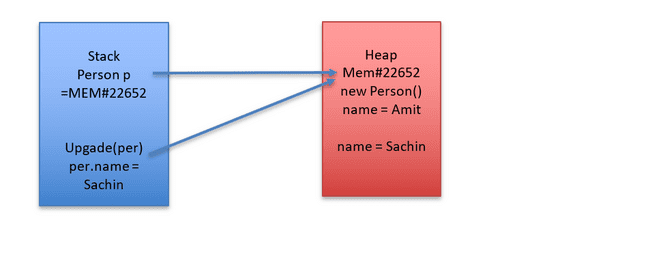
*public class* Test {  
 *static int* i = 1000;  
 *int* j = 1000;  
  
 *public static void* main(String[] args) {  
 Test test = *new* Test();  
 *int* k = 1000;  
 System.out.println(i + "<-->" + test.j + "<-->" + k); ***//1000<-->1000<-->1000*** changeValue(i);  
 changeValue(test.j);  
 changeValue(k);  
 System.out.println(i + "<-->" + test.j + "<-->" + k); ***//1000<-->1000<-->1000 no change in value*** }  
  
 *static void* changeValue(*int* i) {  
 i++;  
 System.out.println(i); ***//1001*** }  
}

But when we pass object we see that the results are different. Here we create an Object of person and pass that object and the value of the object is changing.

*public class* Test {  
  
 *public static void* main(String[] args) {  
 Person p = *new* Person();  
 p.name = "Amit";  
 System.out.println(p); ***//Person{name='Amit'}*** upgrade(p);  
 System.out.println(p); ***//Person{name='Sachin'} value changing*** }  
  
 *static void* upgrade(Person per) {  
 per.name = "Sachin";  
 }  
}  
  
*class* Person {  
 String name;  
  
 *@Override  
 public* String toString() {  
 *return* "Person{" + "name='" + name + '\'' + '}';  
 }  
}

Here the point to notice is that the Person p passed to upgrade() method and not the actual object new Person(). These are 2 different entities linked togather as a reference. Person p is a variable on stack pointing new Person() object on heap.

So when we pass p we are passing a variable p pointing to person. The variable received in upgrade method is unrelated to p in main but that too is point to same object. So when we change p.name. The object on heap i



It will be clear with below example. Here we are passing a reference “aDog” pointing to Dog object from main method to foo. So a new variable gets passed to foo “d”. This is evident from the fact that when we point “d” to new Dog object “aDog” is not changing

*public class* Test {  
  
 *public static void* main(String[] args) {  
 Dog aDog = *new* Dog("Max");  
 Dog oldDog = aDog;  
 foo(aDog);  
 System.out.println(aDog.name.equals("Max")); ***// true*** System.out.println(aDog.name.equals("Fifi")); ***// false*** System.out.println(aDog == oldDog); ***// true*** }  
  
 *public static void* foo(Dog d) {  
 System.out.println(d.name.equals("Max")); ***// true*** d = *new* Dog("Fifi");  
 System.out.println(d.name.equals("Fifi")); ***// true*** }  
}

## Nested Class

It is possible to define a class within another class; such classes are known as nested classes. The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class NestedClass is defined within class Outer, then NestedClass does not exist independently of Outer.

|  |  |
| --- | --- |
| Nested class not assessable from other class will not compile | Is assessable from outer class |
| public class Test {      public static void main(String[] args) {          Outer outer = new Outer();          NestedClass nc = new NestedClass();      }  } | public class Outer {      private int j;      public void outerTest(){          NestedClass nc = new NestedClass();          nc.test();      }      public class NestedClass {          public void test() {              System.out.println(j);          }      }  } |

A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.

So NestedClass can access private members of Main but not reverse.

public class Outer {

    private int j;

    public class NestedClass {

        public void test() {

            System.out.println(j);

        }

    }

}

There are two types of nested classes:

static - A static nested class is one that has the static modifier applied. Because it is static, it must access the non-static members of its enclosing class through an object. That is, it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested classes are seldom used.

non-static (or Inner Class) - An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

Inner class can be initialized in instance methods but not in main method or any other static methods of Outer class.

public class Outer {

    private int j;

    public void outerTest(){

        NestedClass nc = new NestedClass(); *//Will compile*

        nc.test();

    }

    public static void main(String[] args) {

        NestedClass nc = new NestedClass(); *//Won't compile*

    }

    public static void test2(){

        NestedClass nc = new NestedClass(); *//Won't compile*

    }

    public class NestedClass {

        public void test() {

            System.out.println(j);

        }

    }

}

## Object Class



1. **obj.getclass()** 🡪 This implementation of this method in Object class may look like a abstarct class but its not.

*public final native* Class<?> getClass();

The Native implies that the implementation is in platform dependent native code like C or C++. The method is final so can’t be overwritten. It returns an object of class named Class in java. We can use it to get the FCQN, Class name , package etc.

String str = "ddds";  
Class aClass = str.getClass();  
System.out.println(aClass.getName()); ***//FCQN***System.out.println(aClass.getPackage()); ***//Package***System.out.println(aClass.getSimpleName()); ***//Class name***

1. **obj.hashCode()** 🡪 Objective of this method is to create a hash value from the object on which it is applied. The default implementation will creates a hash using the memory location of the object and will return integer. Like getClass this is also native so we can’t see the implementation. We can overwrite the hashcode method but we should follow some guidelines provided by Java in hashCode method documentation.

* If the equals method for 2 objects is returning true then hashcode of 2 objects should be same.
* If the hashCode for 2 objects are same it doesn’t gauranteee that the objects should be same
* For same object in same execution enviornment like same machine, OS etc. the hashcode should be same
* When we are overiding the hashCode() we should also also override the toString() method also and the parameters used for calculating hashCode, same should be used for toString also

Example of overriding hashCode(). Note the hash() method is not from java.lang.Object but java.util.Objects

*class* Person {  
 *public* String name;  
 *public int* age;  
 *public char* gender;  
 *public int* ssn;  
  
 *public* Person(String name, *int* age, *char* gender, *int* ssn) {  
 *this*.name = name;  
 *this*.age = age;  
 *this*.gender = gender;  
 *this*.ssn = ssn;  
 }  
  
 *@Override  
 public int* hashCode() {  
 *return* Objects.hash(name, age, gender, ssn); ***// From util object class*** }

}

With above overritten code, the hashcode will be same for 2 objects with same value. But if we comment out the overridden custom implemntation it will be different which logically doesn’t make sence.

Person person = *new* Person("Sam", 22, 'M',123);  
Person person1 = *new* Person("Sam", 22, 'M',123);  
System.out.println(person.hashCode());  
System.out.println(person1.hashCode());

1. **equals()** 🡪 This method compares the euqality of 2 objects. The default implementation uses == operator so it comared the memory location of 2 objects

*public boolean* equals(Object obj) {  
 *return* (*this* == obj);  
}

We can overide the default behaviour. Some times default behaviour doesn’t make sence, say we have 2 objects of same person with same name, ssn etc. if we create 2 objects using new operator, then equals will treat them as different which is not correct. For example for the same Person example if we use default equals method, it will give false, which is logically not correct.

Person person = *new* Person("Sam", 22, 'M',123);  
Person person1 = *new* Person("Sam", 22, 'M',123);  
System.out.println(person.hashCode()); ***//-1824971834***System.out.println(person1.hashCode()); ***//-1824971834***System.out.println(person1.equals(person)); ***//false which is logically not correct***

We can use IDE like intellij to generate equals

*@Override  
public boolean* equals(Object o) {  
 *if* (o == *null* || getClass() != o.getClass()) *return false*;  
 Person person = (Person) o;  
 *return* age == person.age && gender == person.gender && ssn == person.ssn && Objects.equals(name, person.name);  
}

1. **toString()** 🡪Will return the Sring representation of the Class. Default implementation will give the classname appended with hex code represenatation of the hashcode

*public* String toString() {  
 *return* getClass().getName() + "@" + Integer.toHexString(hashCode());  
}

We can override it to get more readble format

1. **finalize()** 🡪 The Java finalize() method of Object class is a method that the Garbage Collector always calls just before the deletion/destroying the object which is eligible for Garbage Collection to perform clean-up activity This is depricated and it should be left out to JVM for default garbage collection
2. **clone()** 🡪 In Java when we use “=” operator it doesn’t create clone of existing object, rather both the objects refer to same memory location, so when we change one variable other also gets changed. In order to create a clone of an object we have to use clone method provided in Object class which will create a new Object with same properties

*protected native* Object clone() *throws* CloneNotSupportedException;

This method is native and protected. In order to implement this the class has to impelemnt marker interface Clonable and in overiding method just call the super implementation

With cloned object when we check hashCode and equals with default implemtation we see that hashcode is different for 2 objects and the equals also return false confirming that 2 different objects are created and if we change one other will not be changed.

*class* Employee *implements* Cloneable {  
 *public* String employeeId;  
 *public int* salary;  
  
 *public* Employee(String employeeId, *int* salary) {  
 *this*.employeeId = employeeId;  
 *this*.salary = salary;  
 }  
  
 *@Override  
 public* Employee clone() *throws* CloneNotSupportedException {  
 *return* (Employee) *super*.clone();  
 }  
}  
  
  
*public class* Test {  
 *public static void* main(String[] args) *throws* CloneNotSupportedException {  
 Employee employee = *new* Employee("472842", 212232);  
 Employee newEmployee = employee.clone();  
 employee.salary = 0;  
 System.out.println(newEmployee.salary + " <----> " + employee.salary);  
 ***//Changing one employee won't change another*** }  
}

Lets say we have one object having member variable as another Class object. Say Employee has a variable Person so when we create a clone of Employee the original and new object both will point to same person. So changing Person of employee1 will change the peroson of emplyee2 which is clone or vice versa this is shallow cloning

*class* Person{  
 String name;  
  
 *public* Person(String name) {  
 *this*.name = name;  
 }  
}  
  
*class* Employee *implements* Cloneable {  
 *public* String employeeId;  
 *public int* salary;  
 Person person;  
  
 *public* Employee(String employeeId, *int* salary, Person person) {  
 *this*.employeeId = employeeId;  
 *this*.salary = salary;  
 *this*.person = person;  
 }  
  
 *@Override  
 public* Employee clone() *throws* CloneNotSupportedException {  
 *return* (Employee) *super*.clone();  
 }  
}  
  
  
*public class* Test {  
 *public static void* main(String[] args) *throws* CloneNotSupportedException {  
 Person person = *new* Person("SAM");  
 Employee employee = *new* Employee("472842", 212232, person);  
 Employee newEmployee = employee.clone();  
 employee.person.name = "SAMMY";  
 System.out.println(newEmployee.person.name + " <----> " + employee.person.name);  
 ***//Changing one employee won't change another*** }  
}

So here we have to do deep clone by cloning all the indivisula child objects we have to add clone menthod in Person and clone the person attribute of Employee this is deep clone

*class* Person *implements* Cloneable{  
 String name;  
  
 *public* Person(String name) {  
 *this*.name = name;  
 }  
 *@Override  
 public* Person clone() *throws* CloneNotSupportedException {  
 *return* (Person) *super*.clone();  
 }  
}

*public class* Test {  
 *public static void* main(String[] args) *throws* CloneNotSupportedException {  
 Person person = *new* Person("SAM");  
 Employee employee = *new* Employee("472842", 212232, person);  
 Employee newEmployee = employee.clone();  
 newEmployee.person = employee.person.clone();  
 employee.person.name = "SAMMY";  
 System.out.println(newEmployee.person.name + " <----> " + employee.person.name);  
 }  
}

1. wait

# Immutable class and Record

Immutable Objects are those whose state can’t be changed once it is created. Mutable is opposite of this. In Java immutable object are created by keeping the member variables as private final only keeping setter and not keeping any getters. We can mark the class as final also which will prevent any child class from extending it and set the values using constructors

*final class* Employee {  
 *private final* String employeeId;  
 *private final int* salary;  
  
 *public* String getEmployeeId() {  
 *return* employeeId;  
 }  
  
 *public int* getSalary() {  
 *return* salary;  
 }  
  
 *public* Employee(String employeeId, *int* salary) {  
 *this*.employeeId = employeeId;  
 *this*.salary = salary;  
 }  
}

Java 16 gave us an alternative for this by using record class which does the same thing with less boiler plate code

*record* Employee(String employeeId, *int* salary) {  
  
}

Record can have static variables and constructors

*record* Employee(String employeeId, *int* salary) {  
  
 *public static* String company = "ABC";  
  
 Employee(String employeeId, *int* salary) {  
 *this*.employeeId = employeeId;  
 *this*.salary = salary;  
 }}

There is another compact constructor for record

*record* Employee(String employeeId, *int* salary) {  
  
 *public static* String company = "ABC";  
  
 Employee {  
 }}

Record can have static variables and method, instance methods instance variables are in definition. They are final by default. They can implement interface. But they can’t extend any class as all record class by default extend java.lang.Record

# Var

The var keyword in Java, introduced in Java 10, allows for **type inference** of local variables. This means that the compiler can automatically determine the type of a variable based on the value assigned to it . “var” is a reserved identifier name, but not a keyword, which means any existing code that is written before the Java 10 that is, using the var as a variable or a method or a package name, it is not going to be affected. “var“ has to be initialized

# Access Modifiers

There are four types of Java access modifiers, and it is ordered in below order in terms of restrictions with top one being the most restricting and bottom one least retricting.

1. **Private**: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.
2. **Default**: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.
3. **Protected**: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child’s class, it cannot be accessed from outside the package.
4. **Public**: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

# Abstract class

If a class has at least one abstract method must be marked as Abstract class and any class that extend the abstract class must provide the implementation otherwise that class also must be marked as Abstract class

# Inheritance

In Java a class can inherit from another class, like a parent child relation, where a child will have access to all the parent variables and methods except private ones.

*public class* Test {  
 *public static void* main(String[] args) {  
 Animal animal = *new* Animal();  
 animal.eat(); ***// can access parent*** }  
}  
  
*class* Animal{  
 *private* String secret; ***// Animal subclass can't access this private member*** *public void* eat(){  
 System.out.println("Animal eating");  
 }  
}  
  
*class* Dog *extends* Animal{  
  
}

All classes in Java directly or indirectly extend Object class. It is the reason below methods like toString(), hashCode() are available to Car class even though it hasn’t defined it, because these are present in parent Object class.

public class Car {

    public static void main(String[] args) {

        new Car().toString();

        new Car().hashCode();

    }

}

One class can inherit only from one class however multi-level inheritance is allowed which means Toyota extends Car and Car extends object, So Toyota class will have access to all Object methods.

public class Toyota extends Car {

    public static void main(String[] args) {

        Toyota toyota = new Toyota();

        toyota.wheel();

        toyota.toString();

        toyota.hashCode();

    }

}

When child classes are called parent classes are also loaded. In below example when new Toyota () is done Car constructor is also invoked and if we print the memory location then we see that both are at same memory location

public class Test {

    public static void main(String[] args) {

        Toyota toyota = new Toyota();

    }

}

class Car {

    public Car() {

        System.out.println("Car constructor " + this);

    }

}

class Toyota extends Car {

    Toyota() {

        System.out.println("Toyota Constructor " + this);

    }

}

Car constructor Toyota@43d7741f

Toyota Constructor Toyota@43d7741f

If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. So, in case if super class doesn’t have default constructor. Sub class will look to call the default constructor and it will fail

    class Car {

        int wheel;

        public Car(int wheel) {

            this.wheel = wheel;

            System.out.println("Car constructor " + this);

        }

    }

    class Toyota extends Car {

        Toyota(int wheel) {                 //*compilation problem*

            System.out.println("Toyota Constructor " + this);

        }

    }

So, the solution is either have no argument constructor

    class Car {

        int wheel;

        Car(){}     // will solve the compilation problem

        public Car(int wheel) {

            this.wheel = wheel;

            System.out.println("Car constructor " + this);

        }

    }

    class Toyota extends Car {

        Toyota(int wheel) {                 *// will Compile*

            System.out.println("Toyota Constructor " + this);

        }

    }

Or explicitly call the super constructor

    class Car {

        int wheel;

        public Car(int wheel) {

            this.wheel = wheel;

            System.out.println("Car constructor " + this);

        }

    }

    class Toyota extends Car {

        Toyota(int wheel) {

            super(wheel);           // will Compile since we are explicitly calling

            System.out.println("Toyota Constructor " + this);

        }

    }

Say 2 classes are put in same folder without any package structure. It may seem like since they are in same folder, we can keep default access level, but it won’t work since they are not in same package and will give below exception as Toyota class can’t see the constructor.

public class Car {

    Car() {

        System.out.println("Car constructor "+this);

    }

}

Exception in thread "main" java.lang.IllegalAccessError: class Toyota tried to access method 'void Car.<init>()' (Toyota is in unnamed module of

loader com.sun.tools.javac.launcher.Main$MemoryClassLoader @543e710e; Car is in unnamed module of loader 'app')

at Toyota.<init>(Toyota.java:2)

at Toyota.main(Toyota.java:7)

## Upcasting Down casting

**Upcasting** when subclass references super class

*public class* Test {  
 *public static void* main(String[] args) {  
 Animal animal = *new* Dog();  
 animal.eat();  
 }  
}  
  
*class* Animal{  
 *public void* eat(){  
 System.out.println("Animal eating");  
 }  
}  
  
*class* Dog *extends* Animal{  
 *@Override  
 public void* eat(){  
 System.out.println("Dog eating");  
 }  
}

**Down casting**  When super class refers sub class then we use down casting here we need to explicitly do the casting

*public static void* main(String[] args) {  
 Animal an = *new* Dog();  
 Dog dg = (Dog)an;  
 dg.eat();  
}

The underlying object is of Dog type but since it refers to Animal. So, when we try to re-assign to Dog we have to do explicit casting. The casting will work since the underlying object is of type Dog

Directly assigning dog to Animal like below won’t work. The code will compile but will throw **ClassCastException** exception. Since the underlying object created with new is of type Animal

Dog an = (Dog)*new* Animal();

We can prevent such issues with instance of operators where we can do the actual casting after doing instance of check

*public static void* main(String[] args) {  
 Animal an = *new* Dog();  
 *if* (an *instanceof* Dog){  
 Dog dg = (Dog)an;  
 dg.eat();  
 }  
 Dog an1 = *null*;  
 *if* (an1 *instanceof* Dog){  
 an1 = (Dog)*new* Animal();  
 }  
}

The instance of works only for inheritance relationship i.e. the 2 Class getting compared must have parent child relationship. We can’t use is for any random class. This is useful if we want to create a common solution. Say we want to create a common method for different animals. There we can pass Animal reference as method parameters. Then Dog Cat etc. can call these and we can check the type using instance of method and do the operation, otherwise we will have to write separate methods.

*public static void* performAction(Animal animal) {  
 animal.eat();  
 *if*(animal *instanceof* Dog) {  
 Dog dog = (Dog) animal;  
 dog.eat();  
 } *else if* (animal *instanceof* Cat cat) { ***// New syntax since Java 16 where we can pass object also*** cat.eat(); ***// And casting is not needed*** }  
}

## Static Dynamic binding

Inside a Java class will have fields and methods, and these methods and fields can be invoked with the help of class name or with the help of object.

When you try to bring inheritance into your Java project, there will be lot many scenarios of casting where Java needs to determine whether parent class methods/variable needs to be called or child class. There won't be any confusion if you have a single class or when you are invoking static, private and final methods and variables.

Some kind of decision must be taken regarding the variables and methods with inheritance. The effort that is going to be put by your Java to determine which method to invoke or to which variable to invoke is called binding. It is of 2 types

Static 🡪 During compilation time, can be overridden during run time

Dynamic 🡪 During run time

*public class* Test {  
 *public static void* main(String[] args) {  
 Animal animal = *new* Animal();  
 Dog dog = *new* Dog();  
 animal.eat(); ***//Static binding*** dog.eat(); ***//Static binding*** Animal animal1 = *new* Dog();  
 animal1.eat(); ***//First static but during runtimeDynamic binding*** }  
}  
  
*class* Animal{  
 *public void* eat(){  
 System.out.println("Animal eating");  
 }  
}  
  
*class* Dog *extends* Animal{  
 *public void* eat(){  
 System.out.println("Dog eating");  
 }  
}

## Overriding

Overriding is a feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its super-classes or parent classes, so that when the method is invoked using child class reference the child implementation is getting called instead of parent implementation. We can’t override private, static or final methods.

*class* Animal{  
 *public void* eat(){  
 System.out.println("Animal eating");  
 }  
}  
  
*class* Dog *extends* Animal{  
 *@Override* ***// will enforce that the signature is same*** *public void* eat(){  
 System.out.println("Dog eating");  
 }  
}

Below are the rule of overriding is that can be enforced by @Override annotation

1. Only methods inherited from parent can be overwritten.
2. The method signature must be same i.e. same name and parameters.
3. The return type should be either same of the child method return type should be sub class of parent return type
4. Method should not be private, static or final.
5. The access modifier can’t be more restrictive, for e.g. if the super class method is protected then subclass can be public,

It can’t be default or private.

Private (most restrictive) 🡪 default 🡪protected 🡪 public (least restrictive)

1. Child method must not replace or add any new exception. It can add or replace subclass exception. Say parent method throws IOException,

* Child can remove the exception in its implementation completely like elephant class below
* Child can use same IOException like Dog
* Child can replace IOException with FileNotFoundException which is child of IOException
* Child can add FileNotFoundException with IOException
* Child can’t replace or add new exception like SQLException which maintains no hierarchy with IOException e.g. Cat and Lion class below

*class* Animal{  
 *void* eat() *throws* IOException {  
 System.out.println("Animal eating");  
 }  
}  
  
*class* Elephant *extends* Animal{  
 *@Override  
 protected void* eat() { ***// No Exception will compile*** System.out.println("Elephant eating");  
 }  
}  
  
*class* Dog *extends* Animal{  
 *@Override  
 protected void* eat() *throws* IOException { ***//Same exception will compile*** System.out.println("Dog eating");  
 }  
}  
  
*class* Sparrow *extends* Animal{  
 *@Override  
 protected void* eat() *throws* IOException, FileNotFoundException { ***//Same exception and subclass will compile*** System.out.println("Sparrow eating");  
 }  
}  
  
  
  
*class* Ant *extends* Animal{  
 *@Override  
 protected void* eat() *throws* FileNotFoundException { ***//Subclass exception will compile*** System.out.println("Ant eating");  
 }  
}  
  
*class* Cat *extends* Animal{  
 *@Override  
 protected void* eat() *throws* SQLException { ***// Replacing with new exception will not compile*** System.out.println("Cat eating");  
 }  
}  
  
*class* Lion *extends* Animal{  
 *@Override  
 protected void* eat() *throws* IOException, SQLException { ***// Adding new exception will not compile*** System.out.println("Lion eating");  
 }  
}

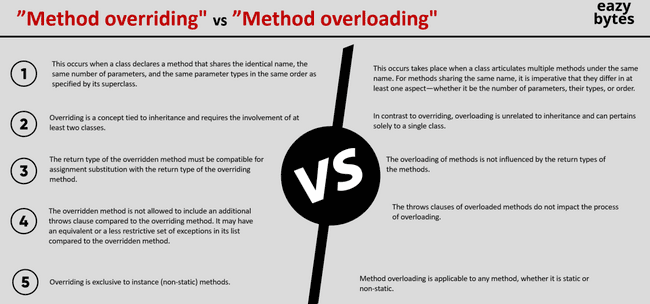
## Super

We can call parent method from child using super key work. If it is 3 level inheritance for e.g. C extending B and B extending A. We can’t call A from C directly.

*class* Animal {  
 *void* eat() {  
 System.out.println("Animal eating");  
 }  
}  
  
*class* Dog *extends* Animal {  
 *@Override  
 void* eat() {  
 *super*.eat();  
 System.out.println("Dog eating");  
 }  
  
 *void* walk(){  
 *super*.eat();  
 }  
}

## Overloading

It is a feature in Java that allows a developer to define multiple methods with the same name in the same class. However, the methods must have different parameters list. Though the method names are same, but they are going to hold the different number of parameters or different data types of parameters, or both of them the return type and the access modifier and the Throws class of a method they are not going to play any role in making an method overloaded, and at the same time using different names for the method parameters does not going to make the method overloaded.



## Final

A field can be declared as final. Doing so prevents its contents from being modified, making it, essentially, a constant. This means that you must initialize a final field when it is declared. You can do this in one of two ways: First, you can give it a value when it is declared. Second, you can assign it a value within a constructor.

    final int breath;  *//won't compile*

    Box(int breath){  //unless initialized in constructor

        this.breath = breath;

    }

Effects of Final

Class 🡪 can’t override

Variable 🡪 Can’t change value

Method 🡪 Can’t be overwritten

## Polymorphism

In Java, Polymorphism refers to the ability to perform a single action in a multiple way. Inheritance allows us to express same class in multiple form. For e.g. a class can be an Dog, Animal, Mammal etc.

Polymorphism in Java can be of 2 type

1. Static compile time 🡪 E.g. overloading
2. Dynamic run time 🡪 E.g. overriding

## Sealed

Inside Java 17, the Sealed classes and interfaces are introduced. With the help of a Sealed class, we can permit which subclass can inherit a superclass.

Syntax

1. Class should use the key word sealed
2. If we want specific class to be allowed need to mention keyword permits
3. Class extending sealed class must be final, sealed or non-sealed

sealed class Animal permits Dog, Cat { **// Only permitted class can extend**}  
  
final class Dog extends Animal { **//Other classes can't extend**}  
  
non-sealed class Cat extends Animal { **//Other classes can extend**}  
  
class Elephant extends Animal{ **//Can't extend Animal as not permitted**}

# Interface

Interface provides a way to define a contract that a class must follow to implement a certain set of behaviors. So, inside an Interface we can define all abstract methods without worrying about the actual implementation logic. All the classes who are trying to implement these Interface, need to adhere to the contract or the specifications that we have defined with the help of abstract methods.

In simple words, an Interface is always going to provide method, signature, return types and parameters that the implementation classes must follow.

With the interface we can provide a standard for our development activity, without this so that each individual developer would follow its own coding standard leading to problems in maintenance of code, difficulty in future enhancements. A great example of interface implementation is JPA, spring security which follows a proper standard , hence when any version is released it is easy to use and understand for developers.

An interface can contain

1. Static final fields 🡪 All field variables are public static and final by default even if we don’t mention it;
2. Abstract methods 🡪 No Implementation abstract keyword is optional, can only be public
3. Default methods
4. Private methods
5. Static methods

The abstract, default and static methods in interface are by default are public. Normally if no modifier is mentioned it is considered as default but for interface its public. We can make separate private methods but abstract, default and static methods are mandated and defaulted to the public.

Since abstract methods by default are public any Class overriding the method must be also public since it can’t have more restrictive access modifier

|  |  |
| --- | --- |
| Interface | Abstract class |
| Can’t make instance | Can make instance by providing implementation |
| Abstract keywork is optional for methods | Abstract keyword is mandatory |
| Interface can have **only abstract** methods. Since Java 8, it can have **default and static methods** also | Abstract class can have abstract and non-abstract methods. |
| Interface can't provide the implementation of abstract class. | Abstract class can provide the implementation of interface. |

One interface can inherit from another interface, in this case we use extends keyword instead of implement keyword as interface won’t have any implementation.

*interface* Vehicle{  
 *void* go();  
  
}  
  
*interface* Car *extends* Vehicle{  
   
}

We can’t create a instance of interface since it doesn’t have any implementation. Only classes can implement an interface, and it has to provide the implementation of all the abstract methods. If the implementing class does not provide any implementation, then that class also should be abstract.

Variables declared in interface are always public, final, static. However, if the interface itself is not public then say it is default, then other classes outside the package won’t be able to see the member variable.

*public class* Test {  
 *public static void* main(String[] args) {  
 System.out.println(Vehicle.TYRES\_COUNT);  
 }  
}  
  
*interface* Vehicle{  
 *int* TYRES\_COUNT = 4;  
 *void* go();  
  
}

After Java 8 Lambda and lots of other features were added then a lot of existing interfaces were modified, and new method signatures were added. This would disrupt all the existing implementations as the new methods had to be implemented by existing implementations. To resolve this default methods were introduced. So, implementing classes can ignore the default, use the default or can override it.

*interface* Car {  
 *default void* startEngine(){  
 System.out.println("VROOM VROOM");  
 }  
}  
  
*class* Toyota *implements* Car{  
 *@Override  
 public void* startEngine(){  
 System.out.println("VROOM VROOM VROOM");  
 }  
}

Since we have implementation logic in default methods hence private methods were introduced to add the common logic at one place in private methods

Similarly static methods were introduced to add utility functions

**Diamond problem with interface**: Interface unlike inheritance allows multiple implements. So, it can face diamond problems

1. Scenario1 both methods are abstract, compilers won’t get confused regarding which method to override. The best approach is to use override annotation

*interface* Vehicle{  
 *void* startEngine();  
  
}  
  
*interface* Car {  
 *void* startEngine();  
}  
  
*class* Toyota *implements* Car , Vehicle{  
 *@Override  
 public void* startEngine() {  
 }  
}

1. Scenario 2 one abstract one default

*interface* Vehicle{  
 *void* startEngine();  
  
}  
  
*interface* Car {  
 *default void* startEngine(){  
 System.out.println("Vroom Vroom");  
 };  
}

In this scenario the compiler will get confused regarding which method to use. The solution is to override the method and then either call parent method or provide new solution

interface Toyota extends Car, Vehicle{  
  
 @Override  
 public default void startEngine() {  
 Car.super.startEngine(); **//or can give its own implementation** }  
}

1. Scenario 3 both interface have default

Here if a class implements only one interface it’s ok it can live with default implementation.

class Toyota implements Car {  
 **//default car method available**}

But if it implements both Car and Vehicle. It must override the method and then can because otherwise compiler will get confused, and it will not compile. There is multiple solution

If the child is another interface, it can just override and keep as abstract class or keep the method as default and provide implementation.

interface Toyota extends Car, Vehicle{  
 @Override  
 public void startEngine();  
}

If the child is another class then then it can call parent default implementation using super or provide new implementation

class Toyota implements Car, Vehicle{  
  
 @Override  
 public void startEngine() {  
 Car.super.startEngine(); **//or can give its own implementation** }  
}

1. If one of them is class and another interface In this case the compiler will give preference to Class as inheritance provides implementation and interface is for specification. So Toyota will have the method from Class Car

*interface* Vehicle {  
 *default void* startEngine() {  
 System.out.println("Vroom");  
 }  
}  
  
*class* Car {  
 *public void* startEngine() {  
 System.out.println("Vroom Vroom");  
 }  
}  
  
*class* Toyota *extends* Car *implements* Vehicle {  
}

\*\* Note if startEngine() method in car is default and not public then it will not compile because then Toyota will inherit the default access modifier of startEngine where as the same method in Vehicle is public(public keywork not needed in interface)

*interface* Vehicle{  
 *default void* startEngine() {  
 System.out.println("Vroom");  
 };  
}  
  
*interface* Car {  
 *default void* startEngine(){  
 System.out.println("Vroom Vroom");  
 };  
}

A **marker interface** is an interface in Java that does not contain any methods, fields, or constants. It is also known as a **tagging interface**. The primary purpose of a marker interface is to provide run-time type information about objects, so the compiler and JVM have additional information about the object. The **Serializable** and **Cloneable** interfaces are the example of marker interface. In short, it indicates a signal or command to the JVM. The declaration of marker interface is the same as interface in Java but the interface must be empty.

Serializable Interface

The **Serializable** interface is a marker interface present in the java.io package. It is used to make an object eligible for saving its state into a file, a process known as **serialization**. Classes that do not implement this interface will not have any of their state serialized or deserialized So in below example if we remove Serializable even if it is not doing anything it will fail

*class* A *implements* Serializable {  
 *int* i;  
 String s;  
  
 *public* A(*int* i, String s) {  
 *this*.i = i;  
 *this*.s = s;  
 }  
}  
  
*public class* Test {  
 *public static void* main(String[] args) *throws* IOException, ClassNotFoundException {  
 A a = *new* A(20, "Amit");  
  
***// Serializing 'a'*** FileOutputStream fos = *new* FileOutputStream("xyz.txt");  
 ObjectOutputStream oos = *new* ObjectOutputStream(fos);  
 oos.writeObject(a);  
  
***// De-serializing 'a'*** FileInputStream fis = *new* FileInputStream("xyz.txt");  
 ObjectInputStream ois = *new* ObjectInputStream(fis);  
 A b = (A) ois.readObject();  
  
 System.out.println(b.i + " " + b.s);  
  
***// closing streams*** oos.close();  
 ois.close();  
 }  
}

Cloneable Interface

The Cloneable interface is present in the java.lang package. It indicates that it is legal for the clone() method to make a field-for-field copy of instances of the class that implements this interface. Invoking the Object.clone() method on an instance of a class that does not implement the Cloneable interface results in a CloneNotSupportedException being thrown

*class* A *implements* Cloneable {  
 *int* i;  
 String s;  
  
 *public* A(*int* i, String s) {  
 *this*.i = i;  
 *this*.s = s;  
 }  
  
 *@Override  
 protected* Object clone() *throws* CloneNotSupportedException {  
 *return super*.clone();  
 }  
}  
  
*public class* Test {  
 *public static void* main(String[] args) *throws* CloneNotSupportedException {  
 A a = *new* A(20, "GeeksForGeeks");  
 A b = (A) a.clone();  
 System.out.println(b.i);  
 System.out.println(b.s);  
 }  
}

Remote Interface

The **Remote** interface is present in the java.rmi package. It marks an object as remote, meaning it can be accessed from another machine. Any object that is a remote object must directly or indirectly implement this interface

Java also allows us to create our own marker interfaces. For example, we can create a marker interface that indicates whether an object can be removed from the database

*interface* Deletable {  
}  
  
*class* Entity *implements* Deletable {  
***// implementation details***}  
  
*public class* Test {  
 *public boolean* delete(Object object) {  
 *if* (!(object *instanceof* Deletable)) {  
 *return false*;  
 }  
 ***// delete implementation details*** *return true*;  
 }  
}

Java provides annotations as an alternative to achieve the same results as marker interfaces. Annotations offer flexible metadata capabilities and can be applied to any class to perform specific actions. However, interfaces allow us to take advantage of polymorphism, which annotations do no

# Array

Array is a group of elements of similar data types. We can store primitives as well as objects in arrays. For e.g., here is a n integer array that can store 5 integer values in it.

        int[] a = {90, 2,7,33,8};

Array in Java are objects. int[] a, here is an object that will point to memory location whose 1st element is 99. Since int takes 4 bytes Java can find the next location easily by knowing next elements. We access arrays using index of elements whhc start at 0 for 1st element and then 1,2,3,4 and so on.

Array declaration can be done in 3 ways

        int[] number;

        int []number1;

        int number2[];

Initialization can be done in below ways

        int[] number = new int[3];

        int []number1 = new int[]{1,2,3};

        int number2[] ={1,2,3};

The size of the array is fixed, which cannot be increased later.

Instead of looping through each element with for we can use For Each. It needs less boilerplate code but the problem is that we can’t get the index related operations as it doesn’t provide the index

        for (int i : number2) {

            System.out.println(i);

        }

Array stores items in sequential order but if we are storing objects then they will be stored in heap in that case the arracy will hold the reference to the Heap object sequentially

Array advantage/Disadvantage



A close up of a text

Description automatically generated

Array methods

*int*[] oldArray = {1,2,3,4,5};  
*int*[] targetArray = Arrays.copyOf(oldArray,3); ***// Array method to create a copy of array****int*[] targetArray1 = Arrays.copyOfRange(oldArray,1, 3);  
  
*for*(*int* num:targetArray1) {  
 System.out.println(num);  
}

*int*[] nums = {1,2,3,4,5};  
System.out.println("Array length "+nums.length); ***// length***

*int*[] smallArray = {5, 2,8,1,6};  
Arrays.sort(smallArray); ***// Array sort***System.out.println(Arrays.toString(smallArray));  
  
*int*[] largerArray = *new int*[10000];  
Random random = *new* Random();  
*for*(*int* i=0;i<largerArray.length;i++){  
 largerArray[i] = random.nextInt(1000000);  
}  
Arrays.parallelSort(largerArray); ***// Use multiple tread can be used for very large arrays***System.out.println(Arrays.toString(largerArray));

*int*[] num = {36, 9, 42, 18, 73};  
System.out.println(Arrays.toString(num));  
Arrays.sort(num);  
System.out.println(Arrays.toString(num));  
*int* index = Arrays.binarySearch(num, 9); ***// Search will work only with sorted array  
//If we comment-out sort part it will return -1 which means not found***System.out.println(index);

# Input Output

We have System.out.println to print to console. Where System class has a variable out which is of type PrintStream which opens a connection to console sends the data and closes it.

Like PrintStream is for output to console there is an InputStream as well which is exposed using System.in

*public static void* main(String[] args) *throws* IOException {  
 System.out.println("Please enter a value");  
 *int* i = System.in.read();  
 System.out.println("User entered --> "+ i);  
}

This will give the ASCII value so 5 will become 53, 7 will become 55 etc. Also, it accepts only single character so if we enter 25 it will take 2

BufferedReader class extends to another Reader class which is an abstract class. BufferReader has additional logic, which is going to read the input data in a byte stream format. This approach can read a single character, a word, a sentence, a file, regardless of whatever amount you are trying to provide as an input. It has a buffer size and it can read that many characters at once making it efficient rather than reading one character at a time. The default is 8 kb as defined in the constructors.

*public* BufferedReader(Reader in) {  
 *this*(in, defaultCharBufferSize);  
}

*private static int* defaultCharBufferSize = 8192;

We can also override the size with another constructor

*public* BufferedReader(Reader in, *int* sz) {  
 *super*(in);  
 *if* (sz <= 0)  
 *throw new* IllegalArgumentException("Buffer size <= 0");  
 *this*.in = in;  
 cb = *new char*[sz];  
 nextChar = nChars = 0;  
}

How to read from console we have to create instance of Reader and pass to BufferedReader constructor which accepts a type of Reader. Since Reader is abstract class we can’t create an instance of it so instead we have to create an instance of its implementations . For console input we have to use InputStreamReader.

*public static void* main(String[] args) *throws* IOException {  
 System.out.println("Please enter a value");  
 InputStreamReader inputStreamReader = *new* InputStreamReader(System.in);  
 BufferedReader bufferedReader = *new* BufferedReader(inputStreamReader);  
 String in = bufferedReader.readLine();  
 System.out.println("User input -->"+ in);  
 bufferedReader.close();  
}

Similarly we can read from file by passing File reader.

FileReader fileReader = *new* FileReader("/Users/eazybytes/Desktop/story.txt");  
BufferedReader reader = *new* BufferedReader(fileReader);  
String line;  
  
*while* ((line = reader.readLine()) != *null*){  
 System.out.println(line);  
}  
reader.close();

There is another convenient way to read input added in Java 1.5

*public static void* main(String[] args) {  
 Scanner sc = *new* Scanner(System.in);  
 System.out.println("Enter your name:");  
 String name = sc.nextLine(); ***// reads the entire input*** System.out.println("Enter your age:");  
 *int* age = sc.nextInt(); ***// Converts to Int*** System.out.println("Hello " + name + " , you are " + age + " years old.");  
 System.out.println("How are you?");  
 String s = sc.next(); ***// reads only one word*** System.out.println(s);  
 sc.close();  
}



Don’t use Print line in for prod

1. Because whenever you want to print a big message onto your console, first it has to get an object of PrintStream. When this object of PrintStream is getting created, a connection to your console will be opened behind the scenes. And with that connection, only the message will be printed onto the console. And at last, the connection with the console, which is established by the PrintStream object, has to be closed behind the scenes, since these kinds of operations are happening behind the scenes, so there is a good chance that a performance degradation can happen.
2. For large projects with large no of classes it will not give information which class and method is printing the logs
3. Can’t disable logging.
4. Can’t maintain logging levels

Logging

*private static final* Logger logger = Logger.getLogger(Test.*class*.getName());  
  
*public static void* main(String[] args) {  
 logger.setLevel(Level.SEVERE);  
 logger.info("This is info level logging"); ***// will keep the message severity as info*** logger.log(Level.WARNING, "This is warning level logging"); ***// Can pass the logging level*** logger.severe("This is severe level logging"); ***// will keep the logging level as severe*** System.out.println("Hello using System.out.println");  
}

# Error & Exception

Exceptions can happen in any system it need not be only because of developer’s mistake. It can also be because any of the expected criteria in missing like Directory missing, Network issue, other system down, Incorrect URL, incorrect requirement provided etc.

Its developer’s responsibility to deal with the exception and log it properly, so that either an alternate approach can be provided, or information is captured properly so that it can be identified what went wrong and proper handling is done.

## Propagation of Exception

In Java when an exception happens it propagates along the execution chain and a stack trace is provided from source to main method calling it. For e.g., in below example error is propagated from method2() till main.

Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()" because "<local0>" is null

at IncorrectCode.method2(Test.java:17)

at IncorrectCode.method1(Test.java:11)

at Test.main(Test.java:4)

*public class* Test {  
 *public static void* main(String[] args) {  
 IncorrectCode.method1();  
 System.out.println("Main"); ***// Will not be executed*** }  
}  
  
*class* IncorrectCode {  
 *public static void* method1() {  
 method2();  
 System.out.println("Method1"); ***// Will not be executed*** }  
  
 *public static void* method2() {  
 String st = *null*;  
 st.length(); ***// Will cause null pointer exception*** System.out.println();  
 System.out.println("Method1");  
 }  
}

In above code when the exception happens it is not handled and the program stops abruptly and nothing after exception gets printed e.g. sysout in method2, method1 and main is not printed. This is because exception is not handled. To complete information we need this information. We can fix these by adding try catch block.

*public class* Test {  
 *public static void* main(String[] args) {  
 IncorrectCode.method1();  
 System.out.println("Main"); ***// will be executed*** }  
}  
  
*class* IncorrectCode {  
 *public static void* method1() {  
 method2();  
 System.out.println("Method1"); ***// will be executed*** }  
  
 *public static void* method2() {  
 String st = *null*;  
 *try* {  
 st.length(); ***// Will cause null pointer exception*** System.out.println("will not print");  
 } *catch* (Exception e) {  
 e.printStackTrace();  
 }  
 System.out.println("Method2");  
 }  
}

Now the program will not end abruptly and will gracefully close and we will get all the information. However line after the exception in try block will not execute **System.out.println("will not print");**

java.lang.NullPointerException: Cannot invoke "String.length()" because "<local0>" is null

at IncorrectCode.method2(Test.java:18)

at IncorrectCode.method1(Test.java:11)

at Test.main(Test.java:4)

at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:77)

at java.base/jdk.internal.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:43)

at java.base/java.lang.reflect.Method.invoke(Method.java:568)

at jdk.compiler/com.sun.tools.javac.launcher.Main.execute(Main.java:419)

at jdk.compiler/com.sun.tools.javac.launcher.Main.run(Main.java:192)

at jdk.compiler/com.sun.tools.javac.launcher.Main.main(Main.java:132)

Method2

Method1

Main

## Multiple Exception

Although catch (Exception ex) can catch all kinds of exception its better to have different exception for different senarios

*private static* Logger logger = Logger.getLogger(  
 Test.*class*.getName());  
  
*public static void* main(String[] args) {  
 String input = *null*;  
 *try* {  
 input = "Super";  
 input = input.toUpperCase();  
 logger.info(input);  
 input = input.substring(1, 10);  
 logger.info(input);  
 } *catch* (NullPointerException ex) { ***// Will catch only the specific exception*** logger.severe("An null pointer exception occurred. Please check your data");  
 } *catch* (ArrayIndexOutOfBoundsException | StringIndexOutOfBoundsException ex) {  
 ***/\*When there is a common logic we can club and the exception must not have parent child relationship  
 Otherwise won't compile. Introduced in Java 7 to reduce duplicate code \*/*** logger.severe("IndexOutOfBoundsException exception occurred. Please check your input data");  
 } *catch* (Exception ex) { ***// Must be at the end as it handles all exception otherwise won't compile*** logger.severe("An exception occurred. Please check your program");  
 }  
}

With multiple exception parent should come after child

*public static void* main(String[] args) {  
 String input = *null*;  
 File myObj = *new* File("filename.txt");  
 *try* {  
 Scanner myReader = *new* Scanner(myObj);  
 } *catch* (IOException e) {  
 System.out.println(e);  
 } *catch* (FileNotFoundException e) { ***// Parent exception class should be below it won't compile*** System.out.println(e);  
 }  
}

## Exception hierarchy in catch block

Exception declaration also must maintain hierarchy with the more specific exception declared first and more generic exception (higher in inheritance hierarchy) declared later. For e.g.in below example since NullPointerException extends RunTimeException which extends Exception class ., hence NullPointerException is declared above Exception in catch block. Exception which are in same level of inheritance hierarchy can be declared before or after, like in the example below ArrayIndexOutOfBoundsException and NullPointerException

public class Test {

    public static void main(String[] args) {

        String st = null;

        try {

            st.length(); // Will cause null pointer exception

            System.out.println("will not print");

        } catch (ArrayIndexOutOfBoundsException ex) {

            ex.printStackTrace(); // can be declared before or after NullPointerException

        } catch (NullPointerException ex) {

            ex.printStackTrace(); // can't be declared after Exception

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

}

## Finally

Sometimes we need to make sure that some housekeeping logic must always execute. Like closing DB connection, closing scanner or file reader otherwise resources are wasted. These things must be placed in finally block so that it executes.

For e.g. In below code even tough the exception is thrown by some other code finally makes sure that scanner is closed.

public class Test {

    public static void main(String[] args) {

        Scanner scanner = null;

        try {

            scanner = new Scanner(System.in);

            int arr[] = { 1, 2, 3 };

            int i = arr[6];

        } catch (Exception e) {

            e.printStackTrace();

        } finally {

            if (scanner != null) {

                scanner.close();

            }

        }

    }

}

Even with a return statement finally gets executed. It is executed before return.

public class Test {

    public static void main(String[] args) {

        System.out.println(method1());

    }

    static String method1 (){

        Scanner scanner = null;

        try {

            scanner = new Scanner(System.in);

            int arr[] = { 1, 2, 3 };

            int i = arr[1];

            return "HI";  // even with return finally will execute

        } catch (Exception e) {

           System.err.println(e);

        } finally {

            if (scanner != null) {

                scanner.close();

                System.out.println("scanner closed");

            }

        }

        return "bye";

    }

}

A try will need at least a catch or a finally. If we don’t put either it will not compile

public static void main(String[] args) {

        Scanner scanner = null;

        try {

            scanner = new Scanner(System.in);

        } finally {

            if (scanner != null) {

                scanner.close();

                System.out.println("scanner closed");

            }

        }

    }

## Try With Resource

For the Classes which implement Callable or AutoCallable we can use trywithResource

Can use multiple resource in try with resource

Can use finally

Resource in try with resources is final can’t be reassigned

*try* (Scanner scanner = *new* Scanner(System.in)) {  
 System.out.println("Enter a number....");  
 *int* number = scanner.nextInt();  
 System.out.println(number);  
} *catch*(Exception ex) {  
 System.out.println("Please provide input in numerical format only and try again...");  
}

## Throws throw

Another way to handle exception is by throwing the exception instead of catching it. With try catch we gobble up the exception and do appropriate handling. But sometimes we are not sure what to do for handling in those scenarios instead of catching the exception we throw the exception and let the calling method handle it. For e.g., if we are writing a utility or a method which will be used differently by different method in those scenarios, we should not handle it but rather throw it and let calling method handle it.

For e.g., in below program in method () we use Thread. Sleep () which throws Interrupted Exception in its definition so in method () we are forced to handle it, either by try catch or by throwing. Here we are throwing it, which means we are forced to handle it in main where we catch it. If we don’t handle there also and throw it would throw the entire stack trace.

public class Test {

    public static void main(String[] args) {

        try {

            method();

        } catch (InterruptedException e) {

            e.printStackTrace();

        }

    }

    static void method() throws InterruptedException{

        Thread.sleep(1900);

    }

}

But if we throw runtime exception, we will not be forced to handle exception. We will see this in exception hierarchy

public class Test {

    public static void main(String[] args) {

        method();

    }

    static void method() throws RuntimeException {

        System.out.println();

    }

}

The throw keyword is used to explicitly throw an exception from a method or any block of code. It can throw both checked and unchecked exceptions

The throws keyword is used in the method signature to declare that a method might throw one or more exceptions. This informs the caller of the method that it needs to handle these exceptions

## Exception Hierarchy

Below is how exception is hierarchy is maintained.



1. **Throwable** is the topmost class in Exception. Extended by Error and Exception class, Errors and exceptions are both types of throwable objects, but they represent different types of problems that can occur during the execution of a program.
2. **Error:-** Errors are usually caused by serious problems that are outside the control of the program, such as running out of memory or a system crash. Errors are represented by the Error class and its subclasses. Some common examples of errors in Java include:

* OutOfMemoryError: Thrown when the Java Virtual Machine (JVM) runs out of memory.
* StackOverflowError: Thrown when the call stack overflows due to too many method invocations.
* NoClassDefFoundError: Thrown when a required class cannot be found.

Since errors are generally caused by problems that cannot be recovered from, it’s usually not appropriate for a program to catch errors. Instead, the best course of action is usually to log the error and exit the program.

1. **Exception:** All exception come under Exception call
2. **RuntimeException:**-> Those exceptions which under RuntimeException are unchecked exception i.e. Java won’t force you to handle it
3. **Checked Exception:**  All exception under Exception which don’t fall under RuntimeException are checked exception i.e. Java will force you to handle it.

## Nested Try block

Try catch can also be nested in that case the exception in inner block will not impact outer block

*try* {  
 System.out.println("Outer try block: Start");  
  
 *try* {  
 System.out.println("Inner try block: Start");  
 ***// Simulating an arithmetic exception*** *int* result = 10 / 0;  
 System.out.println("Inner try block: End");  
 } *catch* (ArithmeticException e) {  
 System.out.println("ArithmeticException caught in inner try block: " + e.getMessage());  
 } *finally* {  
 System.out.println("Inner try block: Finally block");  
 }  
 System.out.println("Outer try block: End");  
} *catch* (ArithmeticException e) {  
 System.out.println("ArithmeticException caught in outer try block: " + e.getMessage());  
} *finally* {  
 System.out.println("Outer try block: Finally block");  
}

## Custom Exception

We can define our own exception to handle our use case. Here we don’t want use to devide by zero hence we are throwing Exception with custom message. Note, since we are throwing Exception we have to handle it, if it was RuntimeException this was not mandatory.

public class Test {

    public static void main(String[] args) {

        try {

            devide(4, 2);

        } catch (Exception e) {

            e.printStackTrace();

        }

    }

    static double devide(int a , int b) throws Exception {

        if (b <=0) {

            throw new Exception("B can't be zero or below zero"+b);

        }

        else return a/b;

    }

}

We can also create our own custom Exception class. This will help in reusability if we want to use in multiple places. Again if it extends Exception it will be checked exception if it extends RuntimeExcetion it will be unchecked exception.

public class Test {

    public static void main(String[] args) {

        try {

            devide(4, 0);

        } catch (Exception e) {

            System.err.println(e);

        }

    }

    static double devide(int a , int b) throws Exception {

        if (b <=0) {

            throw new CustomException("B can't be zero or below zero --> "+b);

        }

        else return a/b;

    }

}

class CustomException extends Exception{

    CustomException(String msg){

        super(msg);

    }

}

# Enumerations

Enumerations are a type in java like class, interface which is used to maintain constants in a specific order. Say we want to store the constants like priority LOW, MEDIUM, HIGH, URGENT and based on the value we want to return SLA in hours. In order to achive this we would have to define constants

public static final int LOW = 7;  
public static final int MEDIUM = 5;  
public static final int HIGH = 3;  
public static final int URGENT = 1;

Instead we can use Enums Enums van be ins

*enum* PriorityEnum {  
 LOW, MEDIUM, HIGH, URGENT  
}  
  
*public class* Test {  
   
 *public static void* main(String[] args) *throws* ParseException {  
 System.out.println(Test.getEstimatedCompletionTime(PriorityEnum.LOW));  
 System.out.println(Test.getEstimatedCompletionTime(PriorityEnum.MEDIUM));  
 System.out.println(Test.getEstimatedCompletionTime(PriorityEnum.HIGH));  
 System.out.println(Test.getEstimatedCompletionTime(PriorityEnum.URGENT));  
 System.out.println(PriorityEnum.URGENT);  
 }  
  
 *public static int* getEstimatedCompletionTime(PriorityEnum priority) {  
 *return switch* (priority) {  
 *case* LOW -> 7;  
 *case* MEDIUM -> 5;  
 *case* HIGH -> 3;  
 *case* URGENT -> 1;  
 };  
 }  
}

We can instantiate Enum like PriorityEnum priorityEnum = null, but can’t create instance. All constants of Enum are by default static and final

Enum can implement another interface, but it can’t implement another Class because all Enum by default implement Enum class. Items of enum have an ordinal associated with it based on the count where it is placed. So 1st element is 1st next 2nd and so on. Enum class extended by all enum provide methods to get Enum values and ordinals

*enum* PriorityEnum *implements* Serializable {  
 LOW, MEDIUM, HIGH, URGENT  
}  
  
*public class* Test {  
  
 *public static void* main(String[] args) *throws* ParseException {  
 System.out.println(PriorityEnum.URGENT);  
 *for* (PriorityEnum priorityEnum: PriorityEnum.values()){  
 System.out.println(priorityEnum.ordinal()); ***// 1, 2, 3...*** System.out.println(priorityEnum.name()); ***//LOW, MEDIUM ...*** }  
 PriorityEnum low = PriorityEnum.valueOf("LOW");  
 System.out.println(low); ***//LOW*** PriorityEnum high = PriorityEnum.values() [2];  
 System.out.println(high); ***//HIGH*** }  
}

Although we can’t create object of ENUM but we can create constructors. The reason is behind the scene JDK during run time creates class for each constants. When we run below code, we can see that the constructor code runs 4 times each for the instance created.

*enum* PriorityEnum *implements* Serializable {  
 LOW, MEDIUM, HIGH, URGENT;  
  
 PriorityEnum() {  
 System.out.println("Hellos" + *this*.ordinal());  
 }  
}  
  
*public class* Test {  
  
 *public static void* main(String[] args) *throws* ParseException {  
 PriorityEnum high = PriorityEnum.HIGH;  
 }  
}

We can associate data with each enum constant by adding instance variables and a constructor to the enum. Here's an example demonstrating how to associate data with enums:

*enum* Day {  
 ***// Each enum constant can have associated data*** SUNDAY("Sun"),  
 MONDAY("Mon"),  
 TUESDAY("Tue"),  
 WEDNESDAY("Wed"),  
 THURSDAY("Thu"),  
 FRIDAY("Fri"),  
 SATURDAY("Sat");  
  
 ***// Instance variable to hold the abbreviation of the day*** *private final* String abbreviation;  
  
 ***// Constructor to initialize the abbreviation*** Day(String abbreviation) {  
 *this*.abbreviation = abbreviation;  
 }  
  
 ***// Getter method to retrieve the abbreviation*** *public* String getAbbreviation() {  
 *return* abbreviation;  
 }  
}  
  
  
*public class* Test {  
  
 *public static void* main(String[] args) {  
 System.out.println("Abbreviations of the days:");  
 *for* (Day day : Day.values()) {  
 System.out.println(day + ": " + day.getAbbreviation());  
 }  
 }  
}

In this example:

* Each enum constant in the Day enum has an associated abbreviation stored as an instance variable.
* The constructor Day(String abbreviation) initializes the abbreviation for each enum constant.
* The getAbbreviation() method allows accessing the abbreviation associated with each enum constant.
* In the Main class, we iterate through all enum constants using Day.values() and print their names along with their associated abbreviations.
* This approach allows you to associate any type of data with each enum constant, providing flexibility and encapsulation within the enum type.

# Collections

The Java collections framework provides various data structures and algorithms implementation to store and retrieve data. This has two main advantages:

We do not have to write code to implement these data structures and algorithms manually.

Our code will be much more efficient as the collections framework is highly optimised.

The Collections in Java is a framework that provides an architecture to store and manipulate the group of objects. Moreover, the collections framework allows us to use a specific data structure for a particular type of data. The Java collections framework provides a set of interfaces and classes to implement various data structures and algorithms.

People often get confused between the collections framework and Collection Interface.

* The Collection interface is the root interface of the collections framework
* Java does not provide direct implementations of the Collection interface but provides implementations of its sub interfaces like List, Set, and Queue
* There is another Collections class in util which provides utility methods



Hierarchy of the Collections framework. The java.util package contains all the classes and interfaces for the Collections framework.





Three interfaces are introduced from Java 21, These three interfaces are SequencedCollection, SequencedSet, and SequencedMap. Reson for introducing 🡪

1. For all the data structures or the collection objects which maintain the order of insertion, they don't have a common parent or a common interface, which can be used by the developers to write some generic code using the data structures that maintain the order of insertion.
2. The second reason why these interfaces are introduced is, in all the data structures or the collection objects, where they maintain the order of insertion, there is no uniform or consistent way to fetch the first element or the last element
3. The third reason is there is no uniform or a proper or a direct way of reversing the elements inside the Collection object, or inside a data structure where the order of insertion is maintained



Since Set and map maintain different types of stucture we SequencedSet and SequencedMap was created along with SequencedCollection. They contain below methods Only reverse is abstarct method rest all are default methods which throws UnsupportedMethodException. Which means if the Class has not provided implementation for this it will throw exception.

SecquenceCollection methods



SequenceSet methods other methods are inherited from SequenceCollection

A black rectangular object with text

Description automatically generated

SequenceMap methods



The Collection interface includes various methods that can be used to perform different operations on objects. These methods are available in all its sub interfaces.

int size();

boolean isEmpty();

boolean contains(Object o);

Iterator<E> iterator();

Object[] toArray();

<T> T[] toArray(T[] a);

default <T> T[] toArray(IntFunction<T[]> generator)

boolean add(E e);

boolean remove(Object o);

boolean containsAll(Collection<?> c);

boolean addAll(Collection<? extends E> c);

boolean removeAll(Collection<?> c);

default boolean removeIf(Predicate<? super E> filter)

boolean retainAll(Collection<?> c);

void clear();

default Spliterator<E> spliterator()

default Stream<E> stream()

default Stream<E> parallelStream()

## List Interface

The List interface is an ordered collection that allows us to store and access elements sequentially. It allows duplicate

Since List is an interface, we cannot create objects from it. To use the functionalities of the List interface, we can use these classes:



List methods



### ArrayList

Array list stores sequential data. An ArrayList is a dynamic array implementation in Java that allows you to create resizable arrays. Unlike regular arrays, ArrayLists can dynamically grow and shrink in size, making them more flexible for handling collections of data. Resizing ArrayList involves creating new arrays and copying elements, which can be costly in terms of memory and performance. However, for most practical purposes, ArrayList provides satisfactory performance.

When to use array list?

1. When needing ordered list based on index.
2. Since it is indexed, accessing any element using index is faster so if we want frequent access it is better.
3. When we don’t care about duplicate.
4. Size of list is changing frequently.
5. Arraylist is great when you want to do a lot of fetch operations since it is index based it fetches result faster. Its not good when you are doing a lot of insert and delete in the middle of List, since any addition or deletion in the middle will need to shift the entire list. Changes at the end are ok since we don’t ned to shift any element. Since most of the apps are heavy on read operation (like add and then read multiple times ) ArrayList is heavily used.

|  |  |  |
| --- | --- | --- |
| initialised Array list  ArrayList<Car> cars = new ArrayList<>(); |  | ArrayList has an ordinary array inside, which acts as a data store. In most cases, we don't specify the exact size of the list. But the internal array must have some size! And so it does. Its default size is 10. |
| cars.add(ferrari); |  | While adding elements the first order of business is to check **whether the internal array has enough space in the internal array** and whether one more element will fit. If there is space, then the new element is added to the end of the list. |
| cars.add(bugatti);  cars.add(lambo);  cars.add(1, ford) |  | if we insert in the middle, i.e. between other elements, first there is a check whether there is enough space in the array. If there is enough space, then **the elements are shifted right**, starting with the position where we're insert the new element. |
| if there are no places to insert new elements into the array. |  | a new array is created inside the ArrayList whose size is the size of the old array times 1.5 plus 1. In our case, the new array's size will be 16. All the current elements will be copied to there immediately.  The old array will be deleted by the garbage collector, and only the new, expanded array will remain. |
| al.remove(3); |  | removing elements makes "holes" in an array. The only way out was **to shift items left** with each removal, and we had to write our own code each time to perform this shift. |

Tips for array

1. don't create an ArrayList with the default size if you're certain it will have at least 100 elements. The internal array

would have to be expanded **6 times** by the time you inserted the 100th element, and all the elements would have to be shifted each time.

from 10 elements to 16

from 16 elements to 25

from 25 to 38

from 38 to 58

from 58 to 88

from 88 to 133 (i.e. size of the old array times 1.5 plus 1)

As you can imagine, this is quite resource-intensive. So, if you already know (even approximately) the required number of items, it's better to create a list with an array of a specific size:

ArrayList<Car> cars = new ArrayList<>(100);

if you add elements more than 100 won’t cause array out of bound exception

1. This strategy also has a flip side. **When you remove objects from an ArrayList, the internal array's size does not decrease automatically.** Suppose we have an ArrayList with a completely full internal array of 88 elements: As the program runs, we remove 77 elements, so only 11 remain: We're only using 11 positions here, but we've allocated memory for 88 elements. That's 8 times more than we need! In this case, we can optimise our memory use with one of the ArrayList class's special methods: **trimToSize()**. This method "trims" the length of the internal array down to the number of elements currently stored in it.
2. To add an element use add(Object obj)
3. To add an element at a specific position use add(i, Object obj).
4. add(int index, E element) inserts item in arrayList shifting each element. For replacing item use set(int index, E element);
5. clear() is more efficient than removeAll() while deleting.
6. add(int index, E element) inserts item in arrayList shifting each element. For replacing item use set(int index, E element);

Accessing eleemnts of List

List items can be accessed in 3 ways

1. Using for loop 🡪 It is simple and effective but we can’t do any index related operation like adding or modifying if we do it with foor loop

*public static void* main(String[] args) *throws* CloneNotSupportedException {  
 List<String> list = *new* ArrayList<>();  
 *for* (*int* i = 0; i < 10; i++) {  
 list.add(FAKER.gameOfThrones().character());  
 }  
  
 *for* (String str : list ){  
 System.out.println(str);  
 }  
}

1. Iterator 🡪 It can be used to read the items in forward direction and we can read and delete items safely without any error. Itr.hasNext() will check for next item and itr.next() will move Iterator to next item.

Its mandatory to call itr.next() otherwise the iterator will not move and give illegal state exception

*public static void* main(String[] args) *throws* CloneNotSupportedException {  
 List<String> list = *new* ArrayList<>();  
 *for* (*int* i = 0; i < 10; i++) {  
 list.add(FAKER.gameOfThrones().character());  
 }  
 System.out.println(list);  
 Iterator<String> itr = list.iterator();  
 *while* (itr.hasNext()){ ***// Check next item***   
 System.out.println(itr.next()); ***// Moves iterator to next item***  
 itr.remove(); ***// Will remove item from list***   
 }  
 System.out.println(list); ***// Empty*** }

1. ListIterator 🡪 It can navigate the items in both forward and backward direaction and can modify items also

hasnext() and next() is for forward

hasPrevious() and previous() is for backward

*public static void* main(String[] args) *throws* CloneNotSupportedException {  
 List<String> list = *new* ArrayList<>();  
 *for* (*int* i = 0; i < 5; i++) {  
 list.add(FAKER.gameOfThrones().character());  
 }  
 ListIterator<String> iterator = list.listIterator();  
 *while* (iterator.hasNext()){  
 System.out.println(iterator.next()); ***// will print in forward directions*** }  
 *while* (iterator.hasPrevious()){ ***// will print in backward directions*** System.out.println(iterator.previous());  
 }}

It can also modify items with set method and also supports adding element

*public static void* main(String[] args) *throws* CloneNotSupportedException {  
 List<String> list = *new* ArrayList<>();  
 *for* (*int* i = 0; i < 5; i++) {  
 list.add(FAKER.gameOfThrones().character());  
 }  
 System.out.println(list); ***// Will print names*** System.out.println(modify(list)); ***//Will print city after modifications  
 //System.out.println(list);***}  
  
*static* List<String> modify(List<String> list){  
 ListIterator<String> iterator = list.listIterator();  
 *while* (iterator.hasNext()){  
 iterator.next();  
 iterator.set(FAKER.gameOfThrones().city()); ***// Will modify the list*** }  
 *return* list;  
}

Altough listIterator provides a lot extra facility, but if we don’t need those features then it is better to stick to Iterator as ListIterator has some performance impact

Modifications of list

*public static void* main(String[] args) *throws* CloneNotSupportedException {  
 List<String> list = *new* ArrayList<>();  
 *for* (*int* i = 0; i < 5; i++) {  
 list.add(FAKER.gameOfThrones().character());  
 }  
 System.out.println(list); ***// Will print GOT character name*** addJunk(list);  
 System.out.println(list); ***// Harry potter character names will get added***}  
  
*static* List<String> addJunk(List<String> lst) {  
 lst.add(FAKER.harryPotter().character());  
 lst.add(FAKER.harryPotter().character());  
 *return* lst;  
}

To prevent this can use immutable list

*public static void* main(String[] args) *throws* CloneNotSupportedException {  
 *var* newList = List.of(FAKER.gameOfThrones().character(),  
 FAKER.gameOfThrones().character(),  
 FAKER.gameOfThrones().character());  
 System.out.println(newList); ***// Will print GOT character name*** addJunk(newList); ***// throw UnsupportedOperationException as List is immutable*** System.out.println(list);  
}  
  
*static* List<String> addJunk(List<String> lst) {  
 lst.add(FAKER.harryPotter().character());  
 lst.add(FAKER.harryPotter().character());  
 *return* lst;  
}

### LinkedList

A linked list is a linear data structure consisting of a sequence of elements, called nodes, where each node contains a data element and a reference (or pointer) to the next node in the sequence. Like array lists, linked lists do not have a fixed size, and the elements are not stored in contiguous memory locations. Instead, each element (node) points to the next element, forming a chain-like structure. Since it implements Deque interface also more details in Queue section

### Vectors

In Java, both ArrayList and Vector implement the List interface and provide the same functionalities. However, there exist some differences between them.

The Vector class synchronises each individual operation. This means whenever we want to perform some operation on vectors, the Vector class automatically applies a lock to that operation.

It is because when one thread is accessing a vector, and at the same time another thread tries to access it, an exception called ConcurrentModificationException is generated. Hence, this continuous use of lock for each operation makes vectors less efficient.

However, in array lists, methods are not synchronised. Instead, it uses the Collections.synchronizedList() method that synchronises the list as a whole.

why hasn't it been removed from Java yet? The fact is that Java professes the principle of backward compatibility. This means that all old code written many years ago will be understood by modern versions of Java.

### Stack

The Java collections framework has a class named Stack that provides the functionality of the stack data structure. The Stack class extends the Vector class. In stack, elements are stored and accessed in Last In First Out manner. That is, elements are added to the top of the stack and removed from the top of the stack. Since Stack extends the Vector class, it inherits all the methods of Vector or List like add, remove, set etc.. Besides these methods, the Stack class includes 5 more methods that distinguish it from Vector.

push() To add an element to the top of the stack, we use the push() method.

pop() To remove an element from the top of the stack, we use the pop() method

peek() The peek() method returns an object from the top of the stack

search() To search an element in the stack, we use the search() method. It returns the position of the element from the top of the stack.

empty() To check whether a stack is empty or not, we use the empty() method.

It is better to use stack specific method rather than List method when applicable

The Stack class provides the direct implementation of the stack data structure. However, it is recommended not to use it. Instead, use the ArrayDeque class (implements the Deque interface) to implement the stack data structure in Java.`

## Map

In Java, elements of Map are stored in key/value pairs. Keys are unique values associated with individual Values. A map cannot contain duplicate keys. And, each key is associated with a single value.



We can access and modify values using the keys associated with them.

In order to use the functionalities of the Map interface, we can use below Classes



The Map interface is also extended by these subinterfaces:



the Map interface also includes the following methods:

put(K, V) - Inserts the association of a key K and a value V into the map. If the key is already present, the new value replaces the old value.

putAll() - Inserts all the entries from the specified map to this map.

putIfAbsent(K, V) - Inserts the association if the key K is not already associated with the value V.

get(K) - Returns the value associated with the specified key K. If the key is not found, it returns null.

getOrDefault(K, defaultValue) - Returns the value associated with the specified key K. If the key is not found, it returns the defaultValue.

containsKey(K) - Checks if the specified key K is present in the map or not.

containsValue(V) - Checks if the specified value V is present in the map or not.

replace(K, V) - Replace the value of the key K with the new specified value V.

replace(K, oldValue, newValue) - Replaces the value of the key K with the new value newValue only if the key K is associated with the value oldValue.

remove(K) - Removes the entry from the map represented by the key K.

remove(K, V) - Removes the entry from the map that has key K associated with value V.

keySet() - Returns a set of all the keys present in a map.

values() - Returns a set of all the values present in a map.

entrySet() - Returns a set of all the key/value mapping present in a map.

Some important items

1. Map is part of Collection framework but not collection interface
2. Map is parameterized with 2 values Map<K, V>
3. Each item of map is an Entry and is denoted by *interface* Entry<K, V> mentioned in side Map interface

### HashMap

It stores elements in key/value pairs. Here, keys are unique identifiers used to associate each value on a map. In Java, elements of Map are stored in key/value pairs. Keys are unique values associated with individual Values. A map cannot contain duplicate keys. And, each key is associated with a single value.

// hashMap creation with 8 capacity and 0.6 load factor

HashMap<K, V> numbers = new HashMap<>();

To add a single element to the hashmap, we use the put(), there is no add method like List interface.

Map<String, String> map = *new* HashMap<>();  
map.put("india", "Delhi");  
map.put("usa", "Washington");  
map.put("australia", "Canberra");

1. Hashmaps store data in an array format where each data element, referred to as a ‘bucket’, holds the key-value pairs.
2. Hash function is critical for determining the index in the array where a particular key-value pair will be stored. The hash function takes a key and returns an integer, which is then used to calculate the index. Hence order is not maintained in Hashmap as order is determined by the index of hash value calculated.
3. Not thread safe. Collections.synchronizedMap is used to make them syscronized.

Map<String, String> map1 = Collections.synchronizedMap(map);

1. In Hashmap collision happen when 2 or more keys produce same hashcode. Ideally key should be unique, but due to finite range of hash codes and potentially infinite no keys, collisions can happen.
2. In case of collision HashMap uses equals method to determine if the Objects are same or not
3. If equals returns true then previous value is removed from the index and the new Object is added at the same index with the same key. So, for e.g. if Obj1 and Obj2 have same hash and true in equals method then the element added later say Obj2 will replace Obj1
4. During collision if the equals method returns false then the 2 objects are considered as different then Map uses linked list to store Entry with same hash in same bucket

Since java 8 improvements have been done to solve hash collision. Linked list are not good for accessing large no of items So when no of collision is more than a specific threshold instead of using LinkedList i.e. if map needs to store more than a specific no of items in a bucket it uses TreeMap

*static final int* TREEIFY\_THRESHOLD = 8;

*static final int* UNTREEIFY\_THRESHOLD = 6;



1. To retrieving items from Map we use get mothod and pass the key map.get("abc"). It generates the hash simmilay way like storing to figure out the bucket where the item is stored to fetch the result. And in case of collision follows the same equals method to eliminate the duplicates



*public static void* main(String[] args) {  
 Map<String, String> map = *new* HashMap<>();  
 map.put("india", "Delhi");  
 map.put("usa", "Washington");  
 map.put("australia", "Canberra");  
 System.out.println(map); ***//{usa=Washington, australia=Canberra, india=Delhi}*** System.out.println("usa".hashCode());  
 System.out.println("australia".hashCode());  
 System.out.println("india".hashCode());  
}

We can retrieve result from Hash map in multiple ways

1. Using keys- We can get a set of keys using keyset() method and iterate over it to get items from map using get method by passing the key

*public static void* approach1(HashMap<String, String> countryMap) {  
 Set<String> keys = countryMap.keySet();  
 Iterator<String> iterator = keys.iterator();  
 *while* (iterator.hasNext()) {  
 String key = iterator.next();  
 String capital = countryMap.get(key);  
 System.out.println(key + " : "+capital);  
 }  
 ***/\*for(String key:keys) {  
 String capital = countryMap.get(key);  
 System.out.println(key + " : "+capital);  
 }\*/***}

1. Using entry- Each item in a HashMap is called Entry defined in a inner interface *interface* Entry<K, v> We can use entrySet() method to retrieve a set of Entry Set<Map.Entry<K, v>>

*public static void* approach2(HashMap<String, String> countryMap) {  
 Set<Map.Entry<String, String>> entries = countryMap.entrySet();  
 Iterator<Map.Entry<String, String>> iterator = entries.iterator();  
 ***/\*while (iterator.hasNext()) {  
 Map.Entry<String,String> entry = iterator.next();  
 String key = entry.getKey();  
 String capital = entry.getValue();  
 System.out.println(key + " : "+capital);  
 }\*/*** *for*(Map.Entry<String, String> entry:entries) {  
 String key = entry.getKey();  
 String capital = entry.getValue();  
 System.out.println(key + " : "+capital);  
 }  
}

1. If we want to process only the values we can use values method

*public static void* approach3(HashMap<String, String> countryMap) {  
 Collection<String> values = countryMap.values();  
 *for*(String value:values) {  
 System.out.println(value);  
 }  
}

We can Change HashMap Value We can use the replace() method to change the value associated with a key in a hashmap.

Remove HashMap Elements To remove elements from a hashmap, we can use the remove() method.

Other Methods of HashMap

1. clear() removes all mappings from the HashMap
2. compute() computes a new value for the specified key
3. computeIfAbsent() computes value if a mapping for the key is not present
4. computeIfPresent() computes a value for mapping if the key is present
5. merge() merges the specified mapping to the HashMap
6. clone() makes the copy of the HashMap
7. containsKey() checks if the specified key is present in Hashmap
8. containsValue() checks if Hashmap contains the specified value
9. size() returns the number of items in HashMap
10. isEmpty() checks if the Hashmap is empty

### SortedMap Interface

The SortedMap interface of the Java collections framework provides sorting of keys stored in a map.

Since SortedMap is an interface, we cannot create objects from it.

In order to use the functionalities of the SortedMap interface, we need to use the class TreeMap that implements it.

Methods of SortedMap

The SortedMap interface includes all the methods of the Map interface. It is because Map is a super interface of SortedMap.

Besides all those methods, here are the methods specific to the SortedMap interface.

comparator() - returns a comparator that can be used to order keys in a map

firstKey() - returns the first key of the sorted map

lastKey() - returns the last key of the sorted map

headMap(key) - returns all the entries of a map whose keys are less than the specified key

tailMap(key) - returns all the entries of a map whose keys are greater than or equal to the specified key

subMap(key1, key2) - returns all the entries of a map whose keys lies in between key1 and key2 including key1

#### NavigableMap Interface

The NavigableMap interface of the Java collections framework provides the features to navigate among the map entries.

It is considered as a type of SortedMap.

In order to use the functionalities of the NavigableMap interface, we need to use the TreeMap class that implements NavigableMap.

The NavigableMap is considered as a type of SortedMap. It is because NavigableMap extends the SortedMap interface.

Hence, all SortedMap methods are also available in NavigableMap. To learn how these methods are defined in SortedMap, visit Java SortedMap.

However, some of the methods of SortedMap (headMap(), tailMap(), and subMap() ) are defined differently in NavigableMap.

headMap(key, booleanValue)

The headMap() method returns all the entries of a navigable map associated with all those keys before the specified key (which is passed as an argument).

The booleanValue is an optional parameter. Its default value is false.

If true is passed as a booleanValue, the method returns all the entries associated with all those keys before the specified key, including the entry associated with the specified key.

tailMap(key, booleanValue)

The tailMap() method returns all the entries of a navigable map associated with all those keys after the specified key (which is passed as an argument) including the entry associated with the specified key.

The booleanValue is an optional parameter. Its default value is true.

If false is passed as a booleanValue, the method returns all the entries associated with those keys after the specified key, without including the entry associated with the specified key.

subMap(k1, bv1, k2, bv2)

The subMap() method returns all the entries associated with keys between k1 and k2 including the entry associated with k1.

The bv1 and bv2 are optional parameters. The default value of bv1 is true and the default value of bv2 is false.

If false is passed as bv1, the method returns all the entries associated with keys between k1 and k2, without including the entry associated with k1.

If true is passed as bv2, the method returns all the entries associated with keys between k1 and k2, including the entry associated with k1.

Other Methods

The NavigableMap provides various methods that can be used to locate the entries of maps.

descendingMap() - reverse the order of entries in a map

descendingKeyMap() - reverses the order of keys in a map

ceilingEntry() - returns an entry with the lowest key among all those entries whose keys are greater than or equal to the specified key

ceilingKey() - returns the lowest key among those keys that are greater than or equal to the specified key

floorEntry() - returns an entry with the highest key among all those entries whose keys are less than or equal to the specified key

floorKey() - returns the highest key among those keys that are less than or equal to the specified key

higherEntry() - returns an entry with the lowest key among all those entries whose keys are greater than the specified key

higherKey() - returns the lowest key among those keys that are greater than the specified key

lowerEntry() - returns an entry with the highest key among all those entries whose keys are less than the specified key

lowerKey() - returns the highest key among those keys that are less than the specified key

firstEntry() - returns the first entry (the entry with the lowest key) of the map

lastEntry() - returns the last entry (the entry with the highest key) of the map

pollFirstEntry() - returns and removes the first entry of the map

pollLastEntry() - returns and removes the last entry of the map

#### TreeMap

The TreeMap class in Java is a part of the Java Collections Framework and provides an efficient way to store key-value pairs in a sorted order based on the natural ordering of its keys or by a custom comparator. It implements the NavigableMap interface. The map is sorted according to the natural ordering of its keys, or by a Comparator provided at map creation time, depending on which constructor is used.

TreeMap 🡪 NavigableMap🡪 SortedMap 🡪 Map

How TreeMap stores data



Insert Elements to TreeMap

put() - inserts the specified key/value mapping (entry) to the map

putAll() - inserts all the entries from specified map to this map

putIfAbsent() - inserts the specified key/value mapping to the map if the specified key is not present in the map

Access TreeMap Elements

1. Using entrySet(), keySet() and values()

entrySet() - returns a set of all the key/values mapping (entry) of a treemap

keySet() - returns a set of all the keys of a tree map

values() - returns a set of all the maps of a tree map

get() - Returns the value associated with the specified key. Returns null if the key is not found.

getOrDefault() - Returns the value associated with the specified key. Returns the specified default value if the key is not found.

Remove TeeMap Elements

remove(key) - returns and removes the entry associated with the specified key from a TreeMap

remove(key, value) - removes the entry from the map only if the specified key is associated with the specified value and returns a boolean value

Replace TreeMap Elements

replace(key, value) - replaces the value mapped by the specified key with the new value

replace(key, old, new) - replaces the old value with the new value only if the old value is already associated with the specified key

replaceAll(function) - replaces each value of the map with the result of the specified function

Methods for Navigation

Since the TreeMap class implements NavigableMap, it provides various methods to navigate over the elements of the treemap.

1. First and Last Methods

firstKey() - returns the first key of the map

firstEntry() - returns the key/value mapping of the first key of the map

lastKey() - returns the last key of the map

lastEntry() - returns the key/value mapping of the last key of the map

Ceiling, Floor, Higher and Lower Methods

higherKey() - Returns the lowest key among those keys that are greater than the specified key.

higherEntry() - Returns an entry associated with a key that is lowest among all those keys greater than the specified key.

lowerKey() - Returns the greatest key among all those keys that are less than the specified key.

lowerEntry() - Returns an entry associated with a key that is greatest among all those keys that are less than the specified key.

ceilingKey() - Returns the lowest key among those keys that are greater than the specified key. If the key passed as an argument is present in the map, it returns that key.

ceilingEntry() - Returns an entry associated with a key that is lowest among those keys that are greater than the specified key. It an entry associated with the key passed an argument is present in the map, it returns the entry associated with that key.

floorKey() - Returns the greatest key among those keys that are less than the specified key. If the key passed as an argument is present, it returns that key.

floorEntry() - Returns an entry associated with a key that is greatest among those keys that are less than the specified key. If the key passed as argument is present, it returns that key.

pollFirstEntry() and pollLastEntry() Methods

pollFirstEntry() - returns and removes the entry associated with the first key of the map

pollLastEntry() - returns and removes the entry associated with the last key of the map

4. headMap(), tailMap() and subMap() Methods

headMap(key, booleanValue)

The headMap() method returns all the key/value pairs of a treemap before the specified key (which is passed as an argument).

The booleanValue parameter is optional. Its default value is false.

If true is passed as a booleanValue, the method also includes the key/value pair of the key which is passed as an argument.

tailMap(key, booleanValue)

The tailMap() method returns all the key/value pairs of a treemap starting from the specified key (which is passed as an argument). The booleanValue is an optional parameter. Its default value is true. If false is passed as a booleanValue, the method doesn't include the key/value pair of the specified key.

subMap(k1, bV1, k2, bV2)

The subMap() method returns all the entries associated with keys between k1 and k2 including the entry of k1. The bV1 and bV2 are optional boolean parameters. The default value of bV1 is true and the default value of bV2 is false. If false is passed as bV1, the method returns all the entries associated with keys between k1 and k2 without including the entry of k1. If true is passed as bV2, the method returns all the entries associated with keys between k1 and k2 including the entry of k2.

### LinkedHashMap

The LinkedHashMap class of the Java collections framework provides the hash table and linked list implementation of the Map interface.

The LinkedHashMap interface extends the HashMap class to store its entries in a hash table. It internally maintains a doubly-linked list among all of its entries to order its entries.

LinkedHashMap Vs. HashMap

Both the LinkedHashMap and the HashMap implements the Map interface. However, there exist some differences between them.

LinkedHashMap maintains a doubly-linked list internally. Due to this, it maintains the insertion order of its elements.

The LinkedHashMap class requires more storage than HashMap. This is because LinkedHashMap maintains linked lists internally.

The performance of LinkedHashMap is slower than HashMap.

### WeakHashMap

The object of a weak reference type can be garbage collected in Java if the reference is no longer used in the program.

Differences Between HashMap and WeakHashMap

Let us see the implementation of a weak hashmap in Java.

import java.util.WeakHashMap;

class Main {

public static void main(String[] args) {

// Creating WeakHashMap of numbers

WeakHashMap<String, Integer> numbers = new WeakHashMap<>();

String two = new String("Two");

Integer twoValue = 2;

String four = new String("Four");

Integer fourValue = 4;

// Inserting elements

numbers.put(two, twoValue);

numbers.put(four, fourValue);

System.out.println("WeakHashMap: " + numbers);

// Make the reference null

two = null;

// Perform garbage collection

System.gc();

System.out.println("WeakHashMap after garbage collection: " + numbers);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

**O**utput

WeakHashMap: {Four=4, Two=2}

WeakHashMap after garbage collection: {Four}

As we can see, when the key two of a weak hashmap is set to null and perform garbage collection, the key is removed.

It is because unlike hashmaps, keys of weak hashmaps are of **weak reference** type. This means the entry of a map are removed by the garbage collector if the key to that entry is no longer used. This is useful to save resources.

Now let us see the same implementation in a hashmap.

import java.util.HashMap;

class Main {

public static void main(String[] args) {

// Creating HashMap of even numbers

HashMap<String, Integer> numbers = new HashMap<>();

String two = new String("Two");

Integer twoValue = 2;

String four = new String("Four");

Integer fourValue = 4;

// Inserting elements

numbers.put(two, twoValue);

numbers.put(four, fourValue);

System.out.println("HashMap: " + numbers);

// Make the reference null

two = null;

// Perform garbage collection

System.gc();

System.out.println("HashMap after garbage collection: " + numbers);

}

}

[Run Code](https://www.programiz.com/java-programming/online-compiler)

Output

HashMap: {Four=4, Two=2}

HashMap after garbage collection: {Four=4, Two=2}

Here, when the key two of the hashmap is set to null and perform garbage collection, the key is not removed.

This is because unlike weak hashmaps keys of hashmaps are of **strong reference** type. This means the entry of a map is not removed by the garbage collector even though the key to that entry is no longer used.

**Note**: All functionalities of hashmaps and weak hashmaps are similar except keys of a weak hashmap are of weak reference, whereas keys of a hashmap are of strong reference.

### EnumMap

The EnumMap class of the Java collections framework provides a map implementation for elements of an enum.

In EnumMap, enum elements are used as keys. It implements the Map interface.

Insert Elements to EnumMap

put() - inserts the specified key/value mapping (entry) to the enum map

putAll() - inserts all the entries of a specified map to this map

For example,

import java.util.EnumMap;

class Main {

enum Size {

SMALL, MEDIUM, LARGE, EXTRALARGE

}

public static void main(String[] args) {

// Creating an EnumMap of the Size enum

EnumMap<Size, Integer> sizes1 = new EnumMap<>(Size.class);

// Using the put() Method

sizes1.put(Size.SMALL, 28);

sizes1.put(Size.MEDIUM, 32);

System.out.println("EnumMap1: " + sizes1);

EnumMap<Size, Integer> sizes2 = new EnumMap<>(Size.class);

// Using the putAll() Method

sizes2.putAll(sizes1);

sizes2.put(Size.LARGE, 36);

System.out.println("EnumMap2: " + sizes2);

}

}

### ConcurrentMap Interface

The ConcurrentMap interface of the Java collections framework provides a thread-safe map. That is, multiple threads can access the map at once without affecting the consistency of entries in a map.

ConcurrentMap is known as a synchronized map.

The ConcurrentMap interface includes all the methods of the Map interface. It is because Map is the super interface of the ConcurrentMap interface.

Besides all those methods, here are the methods specific to the ConcurrentMap interface.

putIfAbsent() - Inserts the specified key/value to the map if the specified key is not already associated with any value.

compute() - Computes an entry (key/value mapping) for the specified key and its previously mapped value.

computeIfAbsent() - Computes a value using the specified function for the specified key if the key is not already mapped with any value.

computeIfPresent() - Computes a new entry (key/value mapping) for the specified key if the key is already mapped with the specified value.

forEach() - Access all entries of a map and perform the specified actions.

merge() - Merges the new specified value with the old value of the specified key if the key is already mapped to a certain value. If the key is not already mapped, the method simply associates the specified value to our key.

#### ConcurrentHashMap

The ConcurrentHashMap class of the Java collections framework provides a thread-safe map. That is, multiple threads can access the map at once without affecting the consistency of entries in a map. It implements the ConcurrentMap interface.

## Set Interface

The Set interface allows us to store elements in different sets like the set in mathematics. It cannot have duplicate elements. Since Set is an interface, we cannot create objects from it. To use functionalities of the Set interface, we can use these classes:



The Set interface is also extended by these sub interfaces:



The Java Set interface allows us to perform basic mathematical set operations like union, intersection, and subset.

* Union - to get the union of two sets x and y, we can use x.addAll(y)
* Intersection - to get the intersection of two sets x and y, we can use x.retainAll(y)
* Subset - to check if x is a subset of y, we can use y.containsAll(x)

Set usually doesn’t maintain order of insertions but some implementations do that like LinkedHashSet. But the primary objective of Set is to remove duplicate element

We can create Set from list which will remove any duplicate items

ArrayList<String> list = *new* ArrayList<>();  
HashSet<String> hashSet = *new* HashSet<>(list);

### HashSet

The HashSet class of the Java Collections framework provides the functionalities of the hash table data structure. HashSet also going to work with the principle of hashing mechanism and hash tables. Whenever we are trying to provide an element to the HashSet,

1. First it is going to calculate the hash code of the element.
2. If there is an element already with the same hash code inside the hash table, then it is going to leverage the equals() method to compare the new element with the existing one, which has the same hash code.
3. If the equals() method returns true, then it means we are trying to add a duplicate element into the Set so it will be discarded.
4. If the equals() method returns false, in such scenarios, the new element is going to be added under the same bucket in an linked list format or in a tree mapper format, because there are multiple elements with the same hash code. So it is going to work very similar to HashMap.

If we see the constructor of HashSet we will see that it internally uses HashMap

*public* HashSet() {  
 map = *new* HashMap<>();  
}

Also, if we see the add method of HashSet it just adds to hashMap with the key as our input value and the Value parameter as a constant called PRESENT

*public boolean* add(E e) {  
 *return* map.put(e, PRESENT)==*null*;  
}



// HashSet with 8 capacity and 0.75 load factor

HashSet<Integer> numbers = new HashSet<>(8, 0.75);

Notice, the part new HashSet<>(8, 0.75). Here, the first parameter is capacity, and the second parameter is loadFactor.

capacity - The capacity of this hash set is 8. Meaning, it can store 8 elements.

loadFactor - The load factor of this hash set is 0.6. This means, whenever our hash set is filled by 60%, the elements are moved to a new hash table of double the size of the original hash table.

By default, the capacity of the hash set will be 16 the load factor will be 0.75

In Java, HashSet is commonly used if we have to access elements randomly. It is because elements in a hash table are accessed using hash codes. The hashcode of an element is a unique identity that helps to identify the element in a hash table. HashSet cannot contain duplicate elements. Hence, each hash set element has a unique hashcode.

Since set always remove duplicates. It relies on the hashCode and equals method to determine unique objects. For custom class we can override hashCode() and equals method to determine unique objects

class Employee {

    int salary;

    int age;

    public Employee(int salary, int age) {

        super();

        this.salary = salary;

        this.age = age;

    }

    @Override

    public int hashCode() {

        return Objects.hash(age);

    }

    @Override

    public boolean equals(Object obj) {

        if (this == obj)

            return true;

        if (obj == null)

            return false;

        if (getClass() != obj.getClass())

            return false;

        Employee other = (Employee) obj;

        return age == other.age;

    }

    @Override

    public String toString() {

        return "Employee [salary=" + salary + ", age=" + age + "]";

    }

}

HashSet don’t have any get method so we rely on Iterator or for each to fetch items

Set<String> superpowers = *new* HashSet<>();superpowers.add("Invisibility");  
superpowers.add("Teleportation");  
superpowers.add("Mind Reading");  
  
*for* (String superPower: superpowers) {  
 System.out.println(superPower.toUpperCase());  
}  
Iterator<String> iterator = superpowers.iterator();  
  
*while* (iterator.hasNext()) {  
 String superPower = iterator.next();  
 System.out.println(superPower.toLowerCase());  
}

Set provides a lot of operations provided by mathematical set like Union (addAll), Intersection(retainAll), Difference(removeAll), Subset(containsAll)

Set<Integer> hashSet1 = *new* HashSet<>(List.of(1, 2, 3, 4, 5));  
Set<Integer> hashSet2 = *new* HashSet<>(List.of(4, 5, 6, 7, 8));  
  
hashSet1.addAll(hashSet2); ***// Union***System.out.println("Union " + hashSet1); ***// Union [1, 2, 3, 4, 5, 6, 7, 8]***hashSet1 = *new* HashSet<>(List.of(1, 2, 3, 4, 5));  
hashSet1.retainAll(hashSet2); ***// Intersection***System.out.println("Intersection" + hashSet1); ***// Intersection[4, 5]***hashSet1 = *new* HashSet<>(List.of(1, 2, 3, 4, 5));  
hashSet1.removeAll(hashSet2); ***// Difference***System.out.println("Difference" + hashSet1); ***// Difference[1, 2, 3]  
  
  
// Subset***hashSet1 = *new* HashSet<>(List.of(1, 2, 3, 4, 5));  
hashSet2 = *new* HashSet<>(List.of(1,2,3));  
*boolean* isSubSet = hashSet1.containsAll(hashSet2);  
System.out.println("Subset " +isSubSet); ***//Subset true***

### Enum Set

The EnumSet class of the Java collections framework provides a set implementation of elements of a single enum. Unlike other set implementations, the enum set does not have public constructors. We must use the predefined methods to create an enum set like allOf , noneOf, range, of

**public** **class** EnumSetImpl {

**enum** Size {

SMALL, MEDIUM, LARGE, EXTRALARGE

}

**public** **static** **void** main(String[] args) {

EnumSet<Size> sizes = EnumSet.*allOf*(Size.**class**);

System.***out***.println("EnumSet: " + sizes);

sizes = EnumSet.*noneOf*(Size.**class**);

System.***out***.println("Empty EnumSet: " + sizes);

sizes = EnumSet.*range*(Size.***MEDIUM***, Size.***EXTRALARGE***);

System.***out***.println("EnumSet: " + sizes);

EnumSet<Size> sizes2 = EnumSet.*of*(Size.***SMALL***, Size.***LARGE***);

System.***out***.println("EnumSet2: " + sizes2);

sizes2.add(Size.***EXTRALARGE***);

}

}

The EnumSet provides an efficient way to store Enum values than other set implementations (like HashSet, TreeSet). An Enum set only stores Enum values of a specific Enum. Hence, the JVM already knows all the possible values of the set. This is the reason why Enum sets are internally implemented as a sequence of bits. Bits specifies whether elements are present in the Enum set or not. The bit of a corresponding element is turned on if that element is present in the set.

### LinkedHashSet

The LinkedHashSet class of the Java collections framework provides functionalities of both the hashtable and the linked list data structure. Elements of LinkedHashSet are stored in hash tables similar to HashSet. However, linked hash sets maintain a doubly-linked list internally for all of its elements. The linked list defines the order in which elements are inserted in hash tables.

// LinkedHashSet with 8 capacity and 0.75 load factor

LinkedHashSet<Integer> numbers = new LinkedHashSet<>(8, 0.75);

// LinkedHashSet with default capacity and load factor

LinkedHashSet<Integer> numbers1 = new LinkedHashSet<>();

LinkedHashSet Vs. HashSet

Both LinkedHashSet and HashSet implements the Set interface. However, there exist some differences between them.

1. LinkedHashSet maintains a linked list internally. Due to this, it maintains the insertion order of its elements.
2. The LinkedHashSet class requires more storage than HashSet. This is because LinkedHashSet maintains linked lists internally.
3. The performance of LinkedHashSet is slower than HashSet. It is because of linked lists present in LinkedHashSet.

### SortedSet

The SortedSet interface of the Java collections framework is used to store elements with some order in a set.In order to use the functionalities of the SortedSet interface, we need to use the TreeSet class that implements it.



The SortedSet interface includes all the methods of the Set interface. It's because Set is a super interface of SortedSet.

Besides methods included in the Set interface, the SortedSet interface also includes these methods:

comparator() - returns a comparator that can be used to order elements in the set

first() - returns the first element of the set

last() - returns the last element of the set

headSet(element) - returns all the elements of the set before the specified element

tailSet(element) - returns all the elements of the set after the specified element including the specified element

subSet(element1, element2) - returns all the elements between the element1 and element2 including element1

#### NavigableSet

The NavigableSet interface of the Java Collections framework provides the features to navigate among the set elements. In order to use the functionalities of the NavigableSet interface, we need to use the TreeSet class that implements NavigableSet.

In order to use the functionalities of the NavigableSet interface, we need to use the TreeSet class that implements NavigableSet.

The NavigableSet is considered as a type of SortedSet. It is because NavigableSet extends the SortedSet interface. Hence, all SortedSet methods are also available in NavigableSet. To learn how these methods, visit Java SortedSet. However, some of the methods of SortedSet (headSet(), tailSet() and subSet()) are defined differently in NavigableSet.

Let's see how these methods are defined in NavigableSet.

**headSet(element, booleanValue)** -The headSet() method returns all the elements of a navigable set before the specified element (which is passed as an argument). The booleanValue parameter is optional. Its default value is false. If true is passed as a booleanValue, the method returns all the elements before the specified element including the specified element.

**tailSet(element, booleanValue)** The tailSet() method returns all the elements of a navigable set after the specified element (which is passed as an argument) including the specified element. The booleanValue parameter is optional. Its default value is true. If false is passed as a booleanValue, the method returns all the elements after the specified element without including the specified element.

**subSet(e1, bv1, e2, bv2)** The subSet() method returns all the elements between e1 and e2 including e1. The bv1 and bv2 are optional parameters. The default value of bv1 is true, and the default value of bv2 is false. If false is passed as bv1, the method returns all the elements between e1 and e2 without including e1. If true is passed as bv2, the method returns all the elements between e1 and e2, including e1.

Methods for Navigation

The NavigableSet provides various methods that can be used to navigate over its elements.

descendingSet() - reverses the order of elements in a set

descendingIterator() - returns an iterator that can be used to iterate over a set in reverse order

ceiling() - returns the lowest element among those elements that are greater than or equal to the specified element

floor() - returns the greatest element among those elements that are less than or equal to the specified element

higher() - returns the lowest element among those elements that are greater than the specified element

lower() - returns the greatest element among those elements that are less than the specified element

pollFirst() - returns and removes the first element from the set

pollLast() - returns and removes the last element from the set

#### TreeSet

The TreeSet class of the Java collections framework provides the functionality of a tree data structure.

In order to create a tree set, we must import the java.util.TreeSet package first. Once we import the package, here is how we can create a TreeSet in Java.

TreeSet<Integer> numbers = new TreeSet<>();

Here, we have created a TreeSet without any arguments. In this case, the elements in TreeSet are sorted naturally (ascending order). However, we can customize the sorting of elements by using the Comparator interface.

Insert Elements to TreeSet

add() - inserts the specified element to the set

addAll() - inserts all the elements of the specified collection to the set

Remove Elements

remove() - removes the specified element from the set

removeAll() - removes all the elements from the set

Methods for Navigation

Since the TreeSet class implements NavigableSet, it provides various methods to navigate over the elements of the tree set.

1. first() and last() Methods

first() - returns the first element of the set

last() - returns the last element of the set

Other methods

higher(element) - Returns the lowest element among those elements that are greater than the specified element.

lower(element) - Returns the greatest element among those elements that are less than the specified element.

ceiling(element) - Returns the lowest element among those elements that are greater than the specified element. If the element passed exists in a tree set, it returns the element passed as an argument.

floor(element) - Returns the greatest element among those elements that are less than the specified element. If the element passed exists in a tree set, it returns the element passed as an argument.

pollFirst() - returns and removes the first element from the set

pollLast() - returns and removes the last element from the set

headSet(element, booleanValue) The headSet() method returns all the elements of a tree set before the specified element (which is passed as an argument). The booleanValue parameter is optional. Its default value is false. If true is passed as a booleanValue, the method returns all the elements before the specified element including the specified element.

tailSet(element, booleanValue) The tailSet() method returns all the elements of a tree set after the specified element (which is passed as a parameter) including the specified element. The booleanValue parameter is optional. Its default value is true. If false is passed as a booleanValue, the method returns all the elements after the specified element without including the specified element.

subSet(e1, bv1, e2, bv2) The subSet() method returns all the elements between e1 and e2 including e1. The bv1 and bv2 are optional parameters. The default value of bv1 is true, and the default value of bv2 is false. If false is passed as bv1, the method returns all the elements between e1 and e2 without including e1. If true is passed as bv2, the method returns all the elements between e1 and e2, including e1.

Set Operations

The methods of the TreeSet class can also be used to perform various set operations.

1. To perform the union between two sets, we use the addAll() method.

TreeSet<Integer> evenNumbers = new TreeSet<>();

evenNumbers.add(2);

evenNumbers.add(4);

System.out.println("TreeSet1: " + evenNumbers);

TreeSet<Integer> numbers = new TreeSet<>();

numbers.add(1);

numbers.add(2);

numbers.add(3);

System.out.println("TreeSet2: " + numbers);

// Union of two sets

numbers.addAll(evenNumbers);

System.out.println("Union is: " + numbers);

1. To perform the intersection between two sets, we use the retainAll() method

TreeSet<Integer> evenNumbers = new TreeSet<>();

evenNumbers.add(2);

evenNumbers.add(4);

System.out.println("TreeSet1: " + evenNumbers);

TreeSet<Integer> numbers = new TreeSet<>();

numbers.add(1);

numbers.add(2);

numbers.add(3);

System.out.println("TreeSet2: " + numbers);

// Intersection of two sets

numbers.retainAll(evenNumbers);

System.out.println("Intersection is: " + numbers);

1. To calculate the difference between the two sets, we can use the removeAll() method.

TreeSet<Integer> evenNumbers = new TreeSet<>();

evenNumbers.add(2);

evenNumbers.add(4);

System.out.println("TreeSet1: " + evenNumbers);

TreeSet<Integer> numbers = new TreeSet<>();

numbers.add(1);

numbers.add(2);

numbers.add(3);

numbers.add(4);

System.out.println("TreeSet2: " + numbers);

// Difference between two sets

numbers.removeAll(evenNumbers);

System.out.println("Difference is: " + numbers);

1. To calculate the difference between the two sets, we can use the removeAll() method.

TreeSet<Integer> numbers = new TreeSet<>();

numbers.add(1);

numbers.add(2);

numbers.add(3);

numbers.add(4);

System.out.println("TreeSet1: " + numbers);

TreeSet<Integer> primeNumbers = new TreeSet<>();

primeNumbers.add(2);

primeNumbers.add(3);

System.out.println("TreeSet2: " + primeNumbers);

// Check if primeNumbers is subset of numbers

boolean result = numbers.containsAll(primeNumbers);

System.out.println("Is TreeSet2 subset of TreeSet1? " + result);

Other Methods of TreeSet

clone() Creates a copy of the TreeSet

contains() Searches the TreeSet for the specified element and returns a boolean result

isEmpty() Checks if the TreeSet is empty

size() Returns the size of the TreeSet

clear() Removes all the elements from the TreeSet

TreeSet Vs. HashSet

Both the TreeSet as well as the HashSet implements the Set interface. However, there exist some differences between them.

* Unlike HashSet, elements in TreeSet are stored in some order. It is because TreeSet implements the SortedSet interface as well.
* TreeSet provides some methods for easy navigation. For example, first(), last(), headSet(), tailSet(), etc. It is because TreeSet also implements the NavigableSet interface.
* HashSet is faster than the TreeSet for basic operations like add, remove, contains and size.

Differences between LinkedHashSet and TreeSet:

* The TreeSet class implements the SortedSet interface. That's why elements in a tree set are sorted. However, the LinkedHashSet class only maintains the insertion order of its elements.
* A TreeSet is usually slower than a LinkedHashSet. It is because whenever an element is added to a TreeSet, it has to perform the sorting operation.
* LinkedHashSet allows the insertion of null values. However, we cannot insert a null value to TreeSet.

TreeSet Comparator

In all the examples above, tree set elements are sorted naturally. However, we can also customize the ordering of elements. For this, we need to create our own comparator class based on which elements in a tree set are sorted

## Queue Interface

The Queue interface of the Java collections framework provides the functionality of the queue data structure. In queues, elements are stored and accessed in First In, First Out manner. That is, elements are added from the behind and removed from the front. Since the Queue is an interface, In order to use the functionalities of Queue, we need to use classes that implement it:





The Queue interface is also extended by various sub interfaces:



Some methods of queue interface

add() - Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.

offer() - Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.

element() - Returns the head of the queue. Throws an exception if the queue is empty.

peek() - Returns the head of the queue. Returns null if the queue is empty.

remove() - Returns and removes the head of the queue. Throws an exception if the queue is empty.

poll() - Returns and removes the head of the queue. Returns null if the queue is empty.

### PriorityQueue

The PriorityQueue class provides the functionality of theheap data structure. Unlike normal queues, priority queue elements are retrieved in sorted order. Suppose, we want to retrieve elements in the ascending order. In this case, the head of the priority queue will be the smallest element. Once this element is retrieved, the next smallest element will be the head of the queue.

It is important to note that the elements of a priority queue may not be sorted. However, elements are always retrieved in sorted order. We can customise the ordering of elements with the help of the Comparator interface.

**Insert** Elements to PriorityQueue

add() - Inserts the specified element to the queue. If the queue is full, it throws an exception.

offer() - Inserts the specified element to the queue. If the queue is full, it returns false.

To access elements from a priority queue, we can use the peek() method. This method returns the head of the queue.

Remove PriorityQueue Elements

remove() - removes the specified element from the queue

poll() - returns and removes the head of the queue

Other PriorityQueue Methods

contains(element) Searches the priority queue for the specified element. If the element is found, it returns true, if not it returns false.

size() Returns the length of the priority queue.

toArray() Converts a priority queue to an array and returns it.

### Deque Interface

The Deque interface of the Java collections framework provides the functionality of a double-ended queue. In a regular queue, elements are added from the rear and removed from the front. However, in a deque, we can **insert and remove elements from both front and rear**.





addFirst() - Adds the specified element at the beginning of the deque. Throws an exception if the deque is full.

addLast() - Adds the specified element at the end of the deque. Throws an exception if the deque is full.

offerFirst() - Adds the specified element at the beginning of the deque. Returns false if the deque is full.

offerLast() - Adds the specified element at the end of the deque. Returns false if the deque is full.

getFirst() - Returns the first element of the deque. Throws an exception if the deque is empty.

getLast() - Returns the last element of the deque. Throws an exception if the deque is empty.

peekFirst() - Returns the first element of the deque. Returns null if the deque is empty.

peekLast() - Returns the last element of the deque. Returns null if the deque is empty.

removeFirst() - Returns and removes the first element of the deque. Throws an exception if the deque is empty.

removeLast() - Returns and removes the last element of the deque. Throws an exception if the deque is empty.

pollFirst() - Returns and removes the first element of the deque. Returns null if the deque is empty.

pollLast() - Returns and removes the last element of the deque. Returns null if the deque is empty

The Stack(LIFO last in first out) class of the Java Collections framework provides the implementation of the stack. However, it is recommended to use Deque as a stack instead of the Stack class. It is because methods of Stack are synchronized. Here are the methods the Deque interface provides to implement stack:

push() - adds an element at the beginning of deque

pop() - removes an element from the beginning of deque

peek() - returns an element from the beginning of deque

#### LinkedList

It provides the functionality of the linked list data structure (doubly linkedlist). E ach element in a linked list is known as a node. It consists of 3 fields:

Prev - stores an address of the previous element in the list. It is null for the first element

Next - stores an address of the next element in the list. It is null for the last element

Data - stores the actual data

Working of a Java LinkedList

Elements in linked lists are not stored in sequence. Instead, they are scattered and connected through links (Prev and Next).



Since the LinkedList class also implements the Queue and the Deque interface, it can implement methods of these interfaces as well. Here are some of the commonly used methods:

Methods Descriptions

addFirst() adds the specified element at the beginning of the linked list

addLast() adds the specified element at the end of the linked list

getFirst() returns the first element

getLast() returns the last element

removeFirst() removes the first element

removeLast() removes the last element

peek() returns the first element (head) of the linked list

poll() returns and removes the first element from the linked list

offer() adds the specified element at the end of the linked list

Both the Java ArrayList and LinkedList implements the List interface of the Collections framework. However, there exists some difference between them.

|  |  |
| --- | --- |
| LinkedList | ArrayList |
| Implements List, Queue, and Deque interfaces. | Implements List interface. |
| Stores 3 values (**previous address**, **data,** and **next address**) in a single position. | Stores a single value in a single position. |
| Provides the doubly-linked list implementation. | Provides a resizable array implementation. |
| Whenever an element is added, prev and next address are changed. | Whenever an element is added, all elements after that position are shifted. |
| To access an element, we need to iterate from the beginning to the element. | Can randomly access elements using indexes. |
|  |  |
|  |  |

#### ArrayDeque

The ArrayDeque class implements these two interfaces: It provide access to both end of the Queue, since it implements Dequeue also

Java Queue Interface

Java Deque Interface



**Add elements** using add(), addFirst() and addLast()

**Insert elements** using offer(), offerFirst() and offerLast()

**Access elements** using getFirst() and getLast()

**Access elements** using peek(), peekFirst() and peekLast() method

peek() - returns the first element of the array deque

peekFirst() - returns the first element of the array deque (equivalent to peek())

peekLast() - returns the last element of the array deque

**Remove elements** using the remove(), removeFirst(), removeLast() method

**Remove elements** using the poll(), pollFirst() and pollLast() method

poll() - returns and removes the first element of the array deque

pollFirst() - returns and removes the first element of the array deque (equivalent to poll())

pollLast() - returns and removes the last element of the array deque

ArrayDeque as a Stack

To implement a LIFO (Last-In-First-Out) stacks in Java, it is recommended to use a deque over the Stack class. The ArrayDeque class is likely to be faster than the Stack class.

ArrayDeque provides the following methods that can be used for implementing a stack.

push() - adds an element to the top of the stack

peek() - returns an element from the top of the stack

pop() - returns and removes an element from the top of the stack

ArrayDeque Vs. LinkedList Class

Both ArrayDeque and Java LinkedList implements the Deque interface. However, there exist some differences between them.

LinkedList supports null elements, whereas ArrayDeque doesn't.

Each node in a linked list includes links to other nodes. That's why LinkedList requires more storage than ArrayDeque.

If you are implementing the queue or the deque data structure, an ArrayDeque is likely to faster than a LinkedList.

### BlockingQueue

BlockingQueue interface of the Java Collections framework extends the Queue interface. It allows any operation to wait until it can be successfully performed. For example, if we want to delete an element from an empty queue, then the blocking queue allows the delete operation to wait until the queue contains some elements to be deleted.

Classes that Implement BlockingQueue

ArrayBlockingQueue

LinkedBlockingQueue

Methods of BlockingQueue

add() - Inserts an element to the blocking queue at the end of the queue. Throws an exception if the queue is full.

element() - Returns the head of the blocking queue. Throws an exception if the queue is empty.

remove() - Removes an element from the blocking queue. Throws an exception if the queue is empty.

offer() - Inserts the specified element to the blocking queue at the end of the queue. Returns false if the queue is full.

peek() - Returns the head of the blocking queue. Returns null if the queue is empty.

poll() - Removes an element from the blocking queue. Returns null if the queue is empty.

The BlockingQueue also provides methods to block the operations and wait if the queue is full or empty.

put() - Inserts an element to the blocking queue. If the queue is full, it will wait until the queue has space to insert an element.

take() - Removes and returns an element from the blocking queue. If the queue is empty, it will wait until the queue has elements to be deleted.

Suppose, we want to insert elements into a queue. If the queue is full then the put() method will wait until the queue has space to insert elements.

Similarly, if we want to delete elements from a queue. If the queue is empty then the take() method will wait until the queue contains elements to be deleted.

Why BlockingQueue?

In Java, BlockingQueue is considered as the thread-safe collection. It is because it can be helpful in multi-threading operations.

Suppose one thread is inserting elements to the queue and another thread is removing elements from the queue.

Now, if the first thread runs slower, then the blocking queue can make the second thread wait until the first thread completes its operation.

#### ArrayBlockingQueue

provides the blocking queue implementation using an array.

put() and take() Method

In multithreading processes, we can use put() and take() to block the operation of one thread to synchronize it with another thread. These methods will wait until they can be successfully executed.

Why use ArrayBlockingQueue?

The ArrayBlockingQueue uses arrays as its internal storage.

It is considered as a thread-safe collection. Hence, it is generally used in multi-threading applications.

Suppose, one thread is inserting elements to the queue and another thread is removing elements from the queue.

Now, if the first thread is slower than the second thread, then the array blocking queue can make the second thread waits until the first thread completes its operations.

#### LinkedBlockingQueue

The LinkedBlockingQueue class of the Java Collections framework provides the blocking queue implementation using a linked list.

Methods of LinkedBlockingQueue

The LinkedBlockingQueue class provides the implementation of all the methods in the BlockingQueue interface. These methods are used to insert, access and delete elements from linked blocking queues. Also, we will learn about two methods put() and take() that support the blocking operation in the linked blocking queue. These two methods distinguish the linked blocking queue from other typical queues.

Insert Elements

add() - Inserts a specified element to the linked blocking queue. It throws an exception if the queue is full.

offer() - Inserts a specified element to the linked blocking queue. It returns false if the queue is full.

Access Elements

peek() - Returns an element from the front of the linked blocking queue. It returns null if the queue is empty.

iterator() - Returns an iterator object to sequentially access an element from the linked blocking queue. It throws an exception if the queue is empty. We must import the java.util.Iterator package to use it.

Remove Elements

remove() - Returns and removes a specified element from the linked blocking queue. It throws an exception if the queue is empty.

poll() - Returns and removes a specified element from the linked blocking queue. It returns null if the queue is empty.

clear() - Removes all the elements from the linked blocking queue.

put() and take() Methods

In multithreading processes, we can use put() and take() to block the operation of one thread to synchronize it with another thread. These methods will wait until they can be successfully executed.

put() Method

To insert the specified element to the end of a linked blocking queue, we use the put() method.

If the linked blocking queue is full, it waits until there is space in the linked blocking queue to insert the element.

take() Method

To return and remove an element from the front of the linked blocking queue, we can use the take() method.

If the linked blocking queue is empty, it waits until there are elements in the linked blocking queue to be deleted.

Why use LinkedBlockingQueue?

The LinkedBlockingQueue uses linked lists as its internal storage.

It is considered as a thread-safe collection. Hence, it is generally used in multi-threading applications.

Suppose, one thread is inserting elements to the queue and another thread is removing elements from the queue.

Now, if the first thread is slower than the second thread, then the linked blocking queue can make the second thread waits until the first thread completes its operations.

## Collection Utility Methods

sort() The sort() method provided by the collections framework is used to sort elements.

shuffle() The shuffle() method of the Java collections framework is used to destroy any kind of order present in the data structure. It does just the opposite of the sorting

reverse() - reverses the order of elements

fill() - replace every element in a collection with the specified value

copy() - creates a copy of elements from the specified source to destination

swap() - swaps the position of two elements in a collection

addAll() - adds all the elements of a collection to other collection

frequency() - returns the count of the number of times an element is present in the collection

disjoint() - checks if two collections contain some common element

## Comparable Comparator

Java provides **Comparable** interface which should be implemented by any custom class if we want to use Arrays or Collections sorting methods. The Comparable interface has compareTo(T obj) method which is used by sorting methods, you can check any Wrapper, String or Date class to confirm this. We should override this method in such a way that it returns a negative integer, zero, or a positive integer if “this” object is less than, equal to, or greater than the object passed as an argument.

For eg we have an Employee object which we want to add to TreeSet. It will throw exception if we don’t implement Comparable.

public class TreeSetImpl {

    public static void main(String[] args) {

        TreeSet<Employee> set = new TreeSet<Employee>();

        set.add(new Employee(50000, 28));

        set.add(new Employee(2333, 33));

        set.add(new Employee(92383, 33));

        set.add(new Employee(7623, 43));

    }

}

class Employee  {

    int salary;

    int age;

    public Employee(int salary, int age) {

        super();

        this.salary = salary;

        this.age = age;

    }

}

Exception in thread "main" java.lang.ClassCastException: class comparableComparator.Employee cannot be cast to class java.lang.Comparable (comparableComparator.Employee is in unnamed module of loader 'app'; java.lang.Comparable is in module java.base of loader 'bootstrap')

at java.base/java.util.TreeMap.compare(TreeMap.java:1291)

at java.base/java.util.TreeMap.put(TreeMap.java:536)

at java.base/java.util.TreeSet.add(TreeSet.java:255)

at comparableComparator.TreeSetImpl.main(TreeSetImpl.java:9)

To Resolve this, we make Employee class to implement Comparable interface and provide implementation for compareTo method

public class TreeSetImpl {

    public static void main(String[] args) {

        TreeSet<Employee> set = new TreeSet<Employee>();

        set.add(new Employee(50000, 28));

        set.add(new Employee(23533, 33));

        set.add(new Employee(92383, 33));

        set.add(new Employee(7623, 43));

        System.out.println(set);

    }

}

class Employee implements Comparable<Employee> {

    int salary;

    int age;

    public Employee(int salary, int age) {

        super();

        this.salary = salary;

        this.age = age;

    }

    @Override

    public int compareTo(Employee arg0) {

        return this.age - arg0.age;

    }

    @Override

    public String toString() {

        return "Employee [salary=" + salary + ", age=" + age + "]";

    }

}

Comparable.compareTo(Object o) method implementation can provide default sorting and we can’t change it dynamically. Whereas with **Comparator**, we can define multiple methods with different ways of sorting and then chose the sorting method based on our requirements.

class SalaryComparator implements Comparator<Employee>{

    @Override

    public int compare(Employee arg0, Employee arg1) {

        return arg0.salary - arg1.salary;

    }

}

        Set<Employee> set1 = new TreeSet<>(new SalaryComparator());

        set1.add(new Employee(50000, 28));

        set1.add(new Employee(23533, 33));

        set1.add(new Employee(92383, 33));

        set1.add(new Employee(7623, 43));

        System.out.println(set1);

|  |  |
| --- | --- |
| **Comparable** make the class implement Comparable interface and provide implementation for **compareTo** method. | **Comparator** create a class which implements Comparator class with implementation for compare method. This class is passed as reference |
| Its fixed and comparison can’t be changed | Since we pass a reference for Comparator we can create multiple Comparator based on different logic |
|  |  |

import java.util.Comparator;

public class StringComparator implements Comparator<String> {

@Override

public int compare(String o1, String o2) {

if (o1.equals(o2)) {

return 0;

} else if (o1.length() < o2.length()) {

return -1;

} else if (o1.length() > o2.length()) {

return 1;

} else {

for (int i = 0; i < o1.length(); i++) {

if (o1.charAt(i) > o2.charAt(i)) {

return 1;

} else {

return -1;

}

}

}

return -1;

}

}

# Generics

Generics was added in Java 5 to provide compile-time type checking and removing risk of ClassCastException that was common while working with collection classes. The whole collection framework was re-written to use generics for type-safety. Let’s see how generics help us using collection classes safely.

    public static void main(String[] args) {

        Integer i =5;

        List list = new ArrayList();

        list.add("abc");

        list.add(i);

        for (Object obj : list) {

            String str = (String) obj;

        }

    }

Above code compiles fine but throws ClassCastException at runtime because we are trying to cast Object in the list to String whereas one of the elements is of type Integer. After Java 5, we use collection classes like below. At the time of list creation, we have specified that the type of elements in the list will be String. So if we try to add any other type of object in the list, the program will throw compile-time error

        List<String> list = new ArrayList();

For example, we have a common Response class which will be returned to User for many scenario. Now if we want to send a response of Employee.

class Response {

    private Object response;

    private String code;

    public Object getResponse() {

        return response;

    }

    public void setResponse(Object response) {

        this.response = response;

    }

    public String getCode() {

        return code;

    }

    public void setCode(String code) {

        this.code = code;

    }

}

public class Test {

    public static void main(String[] args) {

        Employee employee = new Employee(50000, 28);

        Response<Employee> res = new Response();

        res.setResponse(employee);

        res.setCode("200");

    }

}

class Employee {

    int salary;

    int age;

    public Employee(int salary, int age) {

        super();

        this.salary = salary;

        this.age = age;

    }

}

With generics we can do like this

class Response<T> {

    private T response;

    private String code;

    public T getResponse() {

        return response;

    }

    public void setResponse(T response) {

        this.response = response;

    }

    public String getCode() {

        return code;

    }

    public void setCode(String code) {

        this.code = code;

    }

}

    public static void main(String[] args) {

        Employee employee = new Employee(50000, 28);

        Response<Employee> res  = new Response<>();

        res.setResponse(employee);

        res.setCode("200");

        System.out.println(res);

    }

Notice the use of GenericsType class in the main method. We don’t need to do type-casting and we can remove ClassCastException at runtime. If we don’t provide the type at the time of creation, the compiler will produce a warning that “GenericsType is a raw type. References to generic type GenericsType<T> should be parameterized”. When we don’t provide the type, the type becomes Object and hence it’s allowing both String and Integer objects. But, we should always try to avoid this because we will have to use type casting while working on raw type that can produce runtime errors.

Java Generic Type Naming convention helps us understanding code easily and having a naming convention is one of the best practices of Java programming language. So generics also comes with its own naming conventions. Usually, type parameter names are single, uppercase letters to make it easily distinguishable from java variables. The most commonly used type parameter names are:

* E - Element (used extensively by the Java Collections Framework, for example ArrayList, Set etc.)
* K - Key (Used in Map)
* N - Number
* T - Type
* V - Value (Used in Map)
* S,U,V etc. - 2nd, 3rd, 4th types

Sometimes we don’t want the whole class to be parameterized, in that case, we can create java generics method. Since the constructor is a special kind of method, we can use generics type in constructors too. Here is a class showing an example of a java generic method.

    public <T> boolean isEqual(Response<T> g1, Response<T> g2){

        return g1.equals(g2);

    }

Suppose we want to restrict the type of objects that can be used in the parameterized type, for example in a method that compares two objects and we want to make sure that the accepted objects are Comparables. To declare a bounded type parameter, list the type parameter’s name, followed by the extends keyword, followed by its upper bound, similar like below method.

    public static <T extends Comparable<T>> int compare(T t1, T t2){

        return t1.compareTo(t2);

    }

The invocation of these methods is similar to unbounded method except that if we will try to use any class that is not Comparable, it will throw compile-time error. Bounded type parameters can be used with methods as well as classes and interfaces. Java Generics supports multiple bounds also, i.e <T extends A & B & C>. In this case, A can be an interface or class. If A is class then B and C should be an interface. We can’t have more than one class in multiple bounds.

## Covariance

Covariance is a concept in object-oriented programming that allows a subclass to override a method in its superclass with a return type that is a subclass of the return type of the superclass method. This enables more flexibility in method overriding and allows for a more specific return type in the subclass.

In this example, the makeSound() method in the Dog class overrides the same method in the Animal class with a more specific return type (Dog).

*class* Animal {  
 Animal makeSound() {  
 System.out.println("Some sound");  
 *return new* Animal();  
 }  
}  
  
*class* Dog *extends* Animal {  
 *@Override* Dog makeSound() {  
 System.out.println("Woof");  
 *return new* Dog();  
 }  
}  
  
*public class* Test {  
 *public static void* main(String[] args) {  
 Animal animal = *new* Dog();  
 animal.makeSound();  
 }  
}

But same thing is not allowed in Collections

String s = "Generics";  
Object o = s; ***// Will work***  
  
String[] sArray = {"Generics", "Collections"};  
Object[] objArray = sArray; ***// Will work***

List<String> sList = *new* ArrayList<>();  
List<Object> oList = *new* ArrayList<>();  
oList = sList; ***// This will no compile***

This has some disadvantages as we can see we can add String object in Numbers array. Initially in order to achieve polymorphism this covariance was added hence it is found in Arrays but then when the developers resailed it they did not allowed for Collections.

Number[] numArray = {1,2,3};  
Object[] objectArray = numArray; ***// Covariances accepted***objectArray[0] = "String";

## Upper Bound Wildcards

Upper bound wildcards in Java are a feature of generics that allow greater flexibility when working with generic types. They are denoted by the <? extends T> syntax, where T is a specific type.

Upper bound wildcards are used when you want to specify that a parameterized type may be any subtype of a certain type. For instance, if you have a method that accepts a List of any type that is a subtype of Number, you can use an upper bound wildcard to indicate this:

Below method can accept a List<Integer>, List<Double> since these Classes extend java.lang.Number.

*public void* processNumbers(List<? *extends* Number> numbers) {  
 ***// Method implementation***}

Another example using custom class

*class* Employee{}  
  
*class* Developer *extends* Employee{}  
  
  
*public class* Test {  
 *public static void* main(String[] args) {  
 List<Developer> list = *new* ArrayList<>();  
 print(list);  
   
 }  
  
 *static void* print(List<? *extends* Employee> list){  
 *for* (Employee employee: list){  
 System.out.println(employee);  
 }  
 }  
}

When using upper bound wildcards, you can only read from the collection, not write to it. This is because the compiler cannot guarantee the type safety of adding elements to the collection.

So, when we do below it will not compile

*static void* print(List<? *extends* Employee> list){  
 list.add(*new* Developer());  
 *for* (Employee employee: list){  
 System.out.println(employee);  
 }  
}

## Lower Bounded Wildcards

Lower bounded wildcards are denoted by <? super T>, where T is a specific type and we can pass any super class of T. Here we can add item also

*public class* Test {  
 *public static void* main(String[] args) {  
 List<Employee> list = *new* ArrayList<>();  
 list.add(*new* Employee());  
 print(list);  
 }  
  
 *static void* print(List<? *super* Developer> list){  
 *for* (Object employee:list){  
 System.out.println(employee);  
 }  
 }  
}

## Unbounded Wildcards

Unbounded wildcards, denoted by the question mark (?), allow flexibility by accepting any type. They are particularly useful when the code doesn't rely on the specific type of objects in a generic class or method.

List<?>

When the code is agnostic about the type: If the code doesn't depend on the specific type of objects in a generic class or method, unbounded wildcards can be used to provide flexibility.

When working with collections of unknown types: When dealing with collections where the type of elements is unknown or irrelevant, unbounded wildcards can be used to ensure compatibility

# Date Time

Inside Java, there are two different distinct APIs that we can use to represent date and time values.

1. Api in **java.util** and **java.sql** package Legacy API e.g
2. API in **java.time** package introduced in Java 8 as on improvement over legacy API

 A screenshot of a phone

Description automatically generated

Issues with legacy API

1. Ambiguity 🡪 Date represents both date and time. We can't store date and time as a two separate objects types.

Date date = *new* Date();

1. Legacy Date API starts with 1900. So if I want to print 18th December 2024 we have to write like below

Date date = *new* Date(124,11,18);  
System.out.println(date); ***//Wed Dec 18 00:00:00 IST 2024***

So we have to pass 124 for year 1900+ 124 for 2014

11 for December as index starts from 0

Also midnight time is returned by default

1. Mutable not thread safe, new API is immutable hence thread safe.
2. Precision only till milliseconds new date API supports Instant class which supports nanoseconds
3. All the different time zones and daylight savings is not handled properly and is error prone
4. Although different date formats like mm/dd/yy , dd/mm/yyyy etc. is handelled by SimpleDateFormat class in legacy API it is verbose and error prone

## Date

Default implementation of date API returns current date and time. Default implementation calls System class currentTimeMillis method which is a native method which returns the systems time in milliseconds from January 1 1970 epoch time (A fixed date and time used as a reference from which a computer measures system time.)

Default constructor definition

*public* Date() {  
 *this*(System.currentTimeMillis());  
}

@IntrinsicCandidate  
public static native long currentTimeMillis();

Default implementation giving current date and time

Date date = *new* Date();  
System.out.println(date); ***//Wed Dec 18 12:27:45 IST 2024***System.out.println(System.currentTimeMillis()); ***//1734505065137***

For specific dates we need to use a different constructor to which we have to pass Date, month , year time. Legacy Date API starts with 1900. If I want to print 18th December 2024, we must write like below. We must pass 124 for year 1900+ 124 for 2014. 11 for December as index starts from 0 Also midnight is returned by default This constructor is deprecated.

Date date = *new* Date(124,11,18);  
System.out.println(date); ***//Wed Dec 18 00:00:00 IST 2024***

We can use getTime method to get the time in milliseconds.

Date date = *new* Date(124,11,18);  
System.out.println(date); ***//Wed Dec 18 00:00:00 IST 2024***System.out.println(date.getTime()); ***//1734460200000***

If we pass milliseconds to Date constructer we will get the time. Below we are passing current time + 8400000 (1 day ) to get current date + 1day

Date date1 = *new* Date(date.getTime() + 86400000); ***//Date + 1 days***System.out.println(date1); ***//Thu Dec 19 00:00:00 IST 2024***

Can use after method to compare 2 date to check if its greater or not

System.out.println(date.after(date1)); ***//false***System.out.println(date.before(date1)); ***//before***

There is also set method setDate. setMonth etc to set time but these are deprecated we should instead use calender

SQL date

With SQL date we can represent date without time. This was created to support DB columns which support date and not time. If we send date and time both it can cause exception.

Date date = *new* Date(System.currentTimeMillis());

## Simple Date Format

Normal date objects give this format Wed Dec 18 00:00:00 IST 2024. To get a specific format use can use SimpleDateFormat in java.text package. Minute is mm and month is represented as MM.

Date currentDate = *new* Date();  
System.out.println(currentDate); ***// Fri Feb 09 14:52:01 IST 2024***SimpleDateFormat dateFormat = *new* SimpleDateFormat("yy-MMMM-dd HH:mm:ss");  
String formattedDate = dateFormat.format(currentDate);  
System.out.println(formattedDate);  
  
SimpleDateFormat dateFormat1 = *new* SimpleDateFormat("MM/dd/yyyy");  
String formattedDate1 = dateFormat1.format(currentDate);  
System.out.println(formattedDate1);

Sometime we receive time in string, in that case we can parse using Simple Date format

String dateString = "2100-09-26 14:30:00";  
SimpleDateFormat dateFormat2 = *new* SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
Date parsedDate = dateFormat2.parse(dateString);  
System.out.println(parsedDate);

## Calender

To overcome the few of the drawbacks of java.util.Date Java team, they introduced a new set of classes with the name Calendar. This is an abstract class, and this class is implemented by another concrete class which is GregorianCalendar. GregorianCalendar is a hybrid calendar that supports both the Julian and Gregorian calendar systems

The calender has a get instance method which returns Gregorian calender

Calendar calendar = Calendar.getInstance();  
System.out.println(calendar); ***//java.util.GregorianCalendar***

It has a get time method which returns Date object with current time

Calendar calendar = Calendar.getInstance();  
Date date = calendar.getTime();  
System.out.println(date); ***//Thu Dec 19 09:57:33 IST 2024***

We can also create instance of Gregorian calendar, with default constructor we will get current date and time but we can also pass any particular dare and we also don’t gave to do calculation from 1900 like Date class. Month starts with index of zero so January will be 0, February 1 etc., but we can remove the confusion of month by passing constants for month

GregorianCalendar gregorianCalendar = *new* GregorianCalendar(2100, Calendar.JANUARY, 1);  
Date currentDateObj = gregorianCalendar.getTime();  
System.out.println(currentDateObj);

Using Gregorian Calendar object, we can invoke the get() method. To this get() method we can pass a constant which is Calendar.YEAR Calender.Month etc to get specific data. This get method based on the index passed via YEAR, MONTH etc. will call getYear, getMonth etc. but we don’t have to worry about the internal handling

*int* year = gregorianCalendar.get(Calendar.YEAR);  
*int* month = gregorianCalendar.get(Calendar.MONTH);  
*int* date = gregorianCalendar.get(Calendar.DATE);  
*int* hour = gregorianCalendar.get(Calendar.HOUR\_OF\_DAY);  
*int* minutes = gregorianCalendar.get(Calendar.MINUTE);  
*int* seconds = gregorianCalendar.get(Calendar.SECOND);  
*int* milliSeconds = gregorianCalendar.get(Calendar.MILLISECOND);  
  
System.out.printf("Current Date and Time in : %04d-%02d-%02d %02d:%02d:%02d:%04d%n",  
 year, month, date, hour, minutes, seconds,milliSeconds);

Similarly we have set methods also

gregorianCalendar.set(Calendar.YEAR, 2101);  
gregorianCalendar.set(Calendar.MONTH, Calendar.FEBRUARY);  
Date currentDateObj1 = gregorianCalendar.getTime();  
System.out.println(currentDateObj1);

We can format the date using SimpleDateFormat We can also add or subtract date, month etc. from Calender

SimpleDateFormat dateFormat = *new* SimpleDateFormat("yyyy-MMMM-dd HH:mm:ss");  
String formattedDate = dateFormat.format(gregorianCalendar.getTime());  
System.out.println(formattedDate);  
  
gregorianCalendar.add(Calendar.HOUR, 2); ***//2 hours added***gregorianCalendar.add(Calendar.MONTH, -3); ***//3 months reduced***formattedDate = dateFormat.format(gregorianCalendar.getTime());  
System.out.println(formattedDate);

Can also check leap year

*boolean* isLeapYear = gregorianCalendar.isLeapYear(2049);  
System.out.println(isLeapYear);

## Time Zone

Time zone is supported by class TimeZone. Below will list all available time zones

*public static void* main(String[] args) {  
 String[] timeZoneIds = TimeZone.getAvailableIDs();  
  
 *for*(String timeZoneId : timeZoneIds){  
 System.out.println(timeZoneId); ***// Will print all the time zones*** }  
}

Get time zone with date API

SimpleDateFormat dateFormat = *new* SimpleDateFormat("yyyy-MMMM-dd HH:mm:ss");  
dateFormat.setTimeZone(TimeZone.getTimeZone("America/New\_York"));  
String formattedDate = dateFormat.format(*new* Date());  
System.out.println(formattedDate);  
dateFormat.setTimeZone(TimeZone.getTimeZone("Europe/Vatican"));  
formattedDate = dateFormat.format(*new* Date());  
System.out.println(formattedDate)

Get time zone with Gregorian calendar+. We can’t pass getTime method directly as it will local zone time and time zone will be discarded

GregorianCalendar gregorianCalendar = *new* GregorianCalendar(TimeZone.getTimeZone("America/New\_York"));  
*int* year = gregorianCalendar.get(Calendar.YEAR);  
*int* month = gregorianCalendar.get(Calendar.MONTH);  
*int* date = gregorianCalendar.get(Calendar.DATE);  
*int* hour = gregorianCalendar.get(Calendar.HOUR\_OF\_DAY);  
*int* minutes = gregorianCalendar.get(Calendar.MINUTE);  
*int* seconds = gregorianCalendar.get(Calendar.SECOND);  
*int* milliSeconds = gregorianCalendar.get(Calendar.MILLISECOND);  
  
System.out.printf("Current Date and Time in : %04d-%02d-%02d %02d:%02d:%02d:%04d%n",  
 year, month, date, hour, minutes, seconds,milliSeconds);***// New york time***System.out.println(gregorianCalendar.getTime()); ***//This will not give time zone specific time***

## New Date Time

With Java 8 new date time API was introduced which is much more robust. These introduced new classes under java.time package LocalDate, LocalTime, LocalDateTime and ZonedDateTime



Current date time

LocalDate localDate = LocalDate.now();  
LocalTime localTime = LocalTime.now();  
LocalDateTime localDateTime = LocalDateTime.now();  
ZonedDateTime zonedDateTime = ZonedDateTime.now();  
System.out.println(localDate); ***//2024-12-19***System.out.println(localTime); ***//13:35:56.964948500***System.out.println(localDateTime); ***//2024-12-19T13:35:56.964948500***System.out.println(zonedDateTime); ***//2024-12-19T13:35:56.964948500+05:30[Asia/Calcutta]***

**Of Method** To create date and time with specific values we need to use of method

LocalDate date1 = LocalDate.of(2015, 3, 18);  
LocalDate date = LocalDate.of(2015, Month.MARCH, 18); ***// Using Enums for months***LocalTime time = LocalTime.of(14, 30); ***// Can support nanoseconds as well***LocalDateTime dateTime = LocalDateTime.of(2015,3,18,14,30);

ZoneId newYorkTimeZone = ZoneId.of("America/New\_York"); ***//New York time zone***ZonedDateTime zonedDateTime = ZonedDateTime.of(2015,3,18,14,30,0, 0, newYorkTimeZone);  
System.out.println(date);  
System.out.println(time);  
System.out.println(dateTime);  
System.out.println(zonedDateTime);  
  
***//Some of the many utility methods***LocalDate someDate = LocalDate.ofEpochDay(1000); ***//Date of no of days from Epoc date***LocalDate dayIn2000 = LocalDate.ofYearDay(2001, 100); ***// 100th day of year 2001***System.out.println(someDate);  
System.out.println(dayIn2000);

**From method-**  It is a static method to extract date, time

LocalDateTime dateTime = LocalDateTime.of(2015, Month.MARCH, 18, 22, 30);  
System.out.println(dateTime);  
  
LocalDate derivedDate = LocalDate.from(dateTime);  
LocalTime derivedTime = LocalTime.from(dateTime);  
System.out.println(derivedDate);  
System.out.println(derivedTime);

**With methods-** Local date time is immutable. So, when we need to change the date we can use “with” method. It’s not just applicable to LocalDateTime but is applicable to LocalDate, LocalTime etc.

LocalDateTime dateTime = LocalDateTime.of(2015, Month.MARCH, 18, 22, 30);  
System.out.println(dateTime);  
  
LocalDateTime dateTime1 = dateTime.withYear(2030); ***// New Date by modifying year***LocalDateTime dateTime2 = dateTime.withYear(2030).withMonth(10); ***// New date by modifying year and month***System.out.println(dateTime1);  
System.out.println(dateTime2);

**To And At methods** To methods helps to get LocalDate, time etc. from Date time object same we can do with from methods also

LocalDateTime dateTime = LocalDateTime.of(2015, Month.MARCH, 18, 22, 30);  
System.out.println(dateTime);  
  
LocalDate date = dateTime.toLocalDate();  
LocalTime time = dateTime.toLocalTime();  
System.out.println(date);  
System.out.println(time);

**At Methods** will help us to create new date time objects from existing one by supplementing information, this may seem similar to with methods but there is a difference with methods will give same type of object, So date time will give date time object. But with method can give us different type if object for e.g. below we are getting object of LocalDateTime by applying with method to LocalDate

LocalDate localDate = LocalDate.of(2015, 3, 18);  
LocalDateTime startOfDay = localDate.atStartOfDay(); ***// Time will be midnight***System.out.println(startOfDay);  
LocalDateTime localDateTime = localDate.atTime(18, 20); ***// Time will be as passed***System.out.println(localDateTime);

**Get methods** will help us to get specific information like year, month etc.

LocalDate localDate = LocalDate.of(2015, 3 , 18);  
*int* year = localDate.getYear(); ***// 2015***Month month = localDate.getMonth(); ***// MARCH****int* day = localDate.getDayOfMonth();  
DayOfWeek dayOfWeek = localDate.getDayOfWeek(); ***// Monday Tuesday etc***

**Plus minus methods** helps us to do addition subtraction to date time

LocalDate localDate = LocalDate.of(2015, 3 , 18);  
LocalDate ld1 = localDate.plusDays(6); ***// 2015-03-24***LocalDate ld2 = localDate.plusMonths(6); ***// 2015-09-18***LocalDate ld3 = localDate.plusWeeks(2); ***// 2015-04-01***LocalDate ld4 = localDate.minusMonths(9); ***// 2014-06-18***LocalDate ld5 = localDate.minusYears(7); ***// 2008-03-18***System.out.println(ld1);  
System.out.println(ld2);  
System.out.println(ld3);  
System.out.println(ld4);  
System.out.println(ld5);

## Instance

We as humans, we use date and time values, but for machines, the time is calculated based upon the number of seconds passed from the Unix epoch time. What is epoch time? The epoch time inside computers is represented by the time, which is midnight of January 1st, 1970, UTC.



If you try to look at this representation, this zero represents the epoch time, which is midnight of January first, 1970, UTC. After this epoch time, we have various numbers like 1, 2, 3, 4. For each of these numbers, they represent the number of nanoseconds passed from the epoch time. If you try to travel on to the right-hand side, all the nanosecond numbers will be positive. Whereas if you try to travel on to the left-hand side, which is before epoch time, all the numbers are going to be represented with the help of negative numbers.

Each of these numbers inside the timeline, represents an instant time from the epoch time. That is each instance is nano second which is represented by Instant class in Java. Duration class is used to identify the duration of time between two instances.

For example, between 3 and 11 the duration, it is going to be 8 nanoseconds.



Instant can be created like below with now method which will give current instant, but it will give normal Date format representation as toString method is overridden and provides this format

Instant i2 = Instant.now();  
System.out.println(i2); // To String method of Instant will convert to readable format

Instant i1 = Instant.ofEpochSecond(1000);  
System.out.println(i1); ***// Will give time 1000 second from Epoc time****long* seconds = i2.getEpochSecond();  
System.out.println(seconds); ***// No of seconds from Epoc time since the object was created****long* nanoSeconds = i2.getNano(); ***// Not no of nao seconds from epoc time***System.out.println(nanoSeconds);***// No of nanoseconds since the object i2 was created***

Duration cane created like below and it can be added to instant to created new instant

Instant i1 = Instant.ofEpochSecond(1000);  
System.out.println(i1); ***// Will give time 1000 second from Epoc time***Instant i2 = Instant.now();  
System.out.println(i2); ***// To String method of Instant will convert to readable format***Duration d1 = Duration.ofDays(5);  
Duration d2 = Duration.ofMinutes(10);  
Duration d3 = Duration.ofSeconds(30);  
Duration d4 = Duration.ofSeconds(-13);  
Duration d5 = d3.plus(d4);  
  
Instant i3 = i1.plus(d1); ***//add duration to instant***Instant i4 = i2.minus(d2); ***//Subtract duration from instant***System.out.println(i3);  
System.out.println(i4);  
  
*boolean* isAfter = i3.isAfter(i4); ***// false****boolean* isBefore = i3.isBefore(i4); ***// true***Duration duration = Duration.between(i1, i3);  
System.out.println(duration); ***// PT120H 120 hours***

## Period

With the help of Duration, we can identify the duration between two instances. Very similarly, if you are looking to identify the difference between two different dates, we can use Period class. Instant method between method was showing difference in hours while period will show in days, hours minutes

LocalDate localDate1 = LocalDate.of(2013,1,1); ***// 2013-01-01***LocalDate localDate2 = LocalDate.of(2015,3,18); ***// 2015-03-18***Period period = Period.between(localDate1,localDate2); ***// P2Y2M17D 2 years 2 months 17 days***System.out.println(period);  
  
Period oneYearTwoMonths = Period.of(1,2,0); ***// P1Y2M***System.out.println(oneYearTwoMonths);  
Period threeDays = Period.ofDays(3); ***// P3D***LocalDate localDate3 = localDate1.plus(oneYearTwoMonths); ***// 2014-03-01***System.out.println(localDate3); // Can add substract to local date



We can use multiplication, division on Duration and Instant class not applicable for DateTime objects

Duration originalDuration = Duration.ofMinutes(30); ***// PT30M***Duration multipliedDuration = originalDuration.multipliedBy(3); ***// PT1H30M***Duration dividedDuration = originalDuration.dividedBy(2); ***// PT15M***Duration negatedDuration = originalDuration.negated(); ***// PT-30M***Period originalPeriod = Period.ofDays(3); ***// P3D***Period multipliedPeriod = originalPeriod.multipliedBy(3); ***// P9D***Period negatedPeriod = originalPeriod.negated(); ***// P-3D***

Duration always gives duration in Hours Say if we want to truncate the Minutes and seconds we can use truncated

Duration duration = Duration.ofDays(28).  
 plusHours(6).  
 plusMinutes(56).  
 plusSeconds(19); ***// PT678H56M19S***Duration daysTruncated = duration.truncatedTo(ChronoUnit.DAYS); ***// PT672H hours are truncated***Duration hoursTruncated = duration.truncatedTo(ChronoUnit.HOURS); ***// PT678H minutes truncated***Duration minutesTruncated = duration.truncatedTo(ChronoUnit.MINUTES); ***// PT678H56M seconds truncated***

## Zone Offset

Before understanding Zone related class need to understand the concept of Time zone and UTC standard

1. UTC Offset: UTC means **Coordinated Universal Time**. The way it is calculated is that we divide the whole world in multiple 15-degree angle from the centre and the section which has cities like London, Dublin and countries like Denmark, Iceland, Ireland, United Kingdom, Portugal, France, Senegal for UTC 0. From this UTC zero offset if you move in one direction that will be called as UTC+1. And similarly, if we go in other directions it will be UTC -1. India and Sri Lanka are in UTC +5.30.
2. Time Zone: Time zone is going to represent a specific region, A time zone is an area which observes a uniform standard time for legal, commercial and social purposes. Time zones tend to follow the boundaries between countries and their subdivisions instead of strictly following longitude, because it is convenient for areas in frequent communication to keep the same time. Under UTC offset, multiple time zones also can be present, and vice versa single time zone we may have multiple UTC offset.



**ZoneId** is a class inside the new Date-Time API, which represents a specific time zone. We can get the list of time zones using static method getAvailableZoneIds in ZoneId class.

Set<String> allZones = ZoneId.getAvailableZoneIds(); ***// All zones****for* (String zone: allZones){  
 System.out.println(zone);  
}  
ZoneId zone = ZoneId.of("Asia/Kolkata");  
ZoneId destZone = ZoneId.of("America/Chicago");

**ZonedDateTime** class provides date time of a specific zone. By passing this ZoneId to the now () method of ZonedDateTime class, we should be able to get the date of a time zone. We can also use withZoneSameInstant() method if we want to use another time zone time as reference. It will also handle the daylight-saving complexities

ZoneId zone = ZoneId.of("Asia/Kolkata");  
ZoneId destZone = ZoneId.of("America/Chicago");  
  
ZonedDateTime indiaDateTime = ZonedDateTime.now(zone);  
ZonedDateTime chicagoDateTime = indiaDateTime.withZoneSameInstant(destZone);  
System.out.println(indiaDateTime); ***//2024-12-20T22:13:38.450925300+05:30[Asia/Kolkata]***System.out.println(chicagoDateTime); ***//2024-12-20T10:43:38.450925300-06:00[America/Chicago]***

**ZoneOffset** it is going to represent a fixed offset time as per the UTC standard. We should be able to create the ZonedDateTime objects with the help of ZoneOffset as well. Note this will also give us date time as per time zone and not UTC standard. Getting date time from Zone offset is very similar to ZoneID implementation. First, we need to create the objects of ZoneOffset by invoking these ofHours() method and pass the number of the UTC zone whose date time we want. Then we can pass the ZoneOffset to now() and withZoneSameInstant() method we can get ZonedDateTime as per UTC standard

ZoneOffset offset = ZoneOffset.ofHours(2);  
ZoneOffset destOffset = ZoneOffset.ofHours(-8);  
  
ZonedDateTime utc2DateTime = ZonedDateTime.now(offset);  
ZonedDateTime utcDestDateTime = indiaDateTime.withZoneSameInstant(destOffset);  
System.out.println(utc2DateTime); ***//2024-12-20T18:43:38.450925300+02:00***System.out.println(utcDestDateTime); ***//2024-12-20T08:43:38.450925300-08:00***

Both zone id and zone offset will give same information just that zone id date time will have additional zone details

ZoneId zone = ZoneId.of("America/Los\_Angeles");  
ZonedDateTime laZone = ZonedDateTime.now(zone);  
System.out.println(laZone); ***//2024-12-20T19:08:47.273808500-08:00[America/Los\_Angeles]***ZoneOffset laOffset = ZoneOffset.ofHours(-8);  
ZonedDateTime laDateTime = ZonedDateTime.now(laOffset);  
System.out.println(laDateTime); ***//2024-12-20T19:08:47.273808500-08:00***

**OffsetDateTime-** If you are simply looking for an option which is going to help you to build the date and time based upon the offset, ignoring all the daylight-saving rules and geographical locations in such scenarios, you can happily use OffsetDateTime.  For this we can only use ZoneOffset object, but not the ZoneId objects

ZoneOffset offset = ZoneOffset.ofHours(2);  
ZoneOffset destOffset = ZoneOffset.ofHours(-8);  
  
OffsetDateTime offsetDateTime = OffsetDateTime.now(offset);  
OffsetDateTime destOffsetDateTime = offsetDateTime.withOffsetSameInstant(destOffset);  
System.out.println(offsetDateTime);  
System.out.println(destOffsetDateTime);

ZoneId = ZoneOffset + ZoneRules

|  |  |  |
| --- | --- | --- |
| ZoneId zone = ZoneId.of("America/Los\_Angeles"); ZonedDateTime laZone = ZonedDateTime.now(zone); System.out.println(laZone); | ZoneOffset laOffset = ZoneOffset.ofHours(-8); ZonedDateTime laDateTime = ZonedDateTime.now(laOffset); System.out.println(laDateTime); | ZoneOffset laOffset = ZoneOffset.ofHours(-8); OffsetDateTime destOffsetDateTime = OffsetDateTime.now(laOffset); System.out.println(destOffsetDateTime); |

We have another class OffsetTime to represent time

OffsetTime offsetTime = OffsetTime.of(15, 30, 0, 0, ZoneOffset.ofHours(2));  
System.out.println(offsetTime);  
*int* hour = offsetTime.getHour();  
*int* minutes = offsetTime.getMinute();  
*int* second = offsetTime.getSecond();  
ZoneOffset offset = offsetTime.getOffset();  
OffsetTime laterTime = offsetTime.plusHours(2);  
*boolean* isBefore = offsetTime.isBefore(laterTime);

DST

Daylight Saving Time is a concept where few countries like United States, Canada and Australia, they are going to advance their clocks by one hour during the warmer months so that darkness falls at a later clock time. If you take United States as an example, what they are going to do is, they are going to advance their clock by one hour on every March 2nd Sunday at midnight, 2 a.m. Similarly during the November 1st Sunday after the midnight, they're going to make their clock one hour behind.

Lets take example of a fight taking off on 8th march from LA it will reach NY at 5 am

ZonedDateTime departure1 = ZonedDateTime.of(2024, 3, 8, 21, 0, 0, 0, ZoneId.of("America/Los\_Angeles"));  
***// Expected arrival in NYC (assuming flight duration is 4 hours)***ZonedDateTime arrival1 = departure1.plusHours(4).withZoneSameInstant(ZoneId.of("America/New\_York"));  
System.out.println("Departure (PST): " + departure1); ***//Departure (PST): 2024-03-08T21:00-08:00[America/Los\_Angeles]***System.out.println("Arrival (EST): " + arrival1); ***//Arrival (EST): 2024-03-09T04:00-05:00[America/New\_York]***

Same flight taking off on 9th March after day light savings will reach at 5 a.m

ZonedDateTime departure = ZonedDateTime.of(2024, 3, 9, 21, 0, 0, 0, ZoneId.of("America/Los\_Angeles"));  
***// Expected arrival in NYC (assuming flight duration is 4 hours)***ZonedDateTime arrival = departure.plusHours(4).withZoneSameInstant(ZoneId.of("America/New\_York"));  
System.out.println("Departure (PST): " + departure); ***//Departure (PST): 2024-03-09T21:00-08:00[America/Los\_Angeles]***System.out.println("Arrival (EST): " + arrival); ***//Arrival (EST): 2024-03-10T05:00-04:00[America/New\_York]***

So zoned date will take care of such daylight savings as well

Most of the countries in the world they follow the GregorianCalendar. We call this GregorianCalendar as ISO calendar or English calendar, but apart from ISO or GregorianCalendar, we have other types of non-ISO calendars as well exist in few countries. Such calendars are Thai Buddhist calendar, which is going to be followed inside Thailand. Hijra calendar which is going to be followed by the Islamic countries. Minguo calendar, which is followed inside the China and Japanese calendar, which is followed inside Japan. All these kinds of calendars we can call as non-ISO calendars.  We can also convert a non ISO date to ISO date and vice versa.

JapaneseDate japaneseDate = JapaneseDate.now();  
LocalDate localDate = LocalDate.now();  
System.out.println(japaneseDate);  
System.out.println(localDate);  
  
JapaneseDate japaneseDate1 = JapaneseDate.from(localDate);  
LocalDate localDate1 = LocalDate.from(japaneseDate);  
System.out.println(japaneseDate1);  
System.out.println(localDate1);

We have DateTimeFormatter class to format the dates with new API

LocalDate localDate = LocalDate.of(2015,3,18);  
LocalTime localTime = LocalTime.of(15,30,0);  
LocalDateTime localDateTime = LocalDateTime.of(2015,3,18,15,30,0);  
System.out.println(localDate); ***//2015-03-18***System.out.println(localTime); ***//15:30***System.out.println(localDateTime); ***//2015-03-18T15:30***DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("yyyy/MM/dd");  
String formattedDate = dateFormatter.format(localDate);  
System.out.println(formattedDate); ***//2015/03/18***DateTimeFormatter timeFormatter = DateTimeFormatter.ofPattern("h:mm a");  
String formattedTime = timeFormatter.format(localTime);  
System.out.println(formattedTime); ***//3:30 PM***DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("dd/MM/yyyy h:mm a");  
String formattedDateTime = dateTimeFormatter.format(localDateTime);  
System.out.println(formattedDateTime); ***//18/03/2015 3:30 PM***String formattedDate1 = localDate.format(DateTimeFormatter.ISO\_LOCAL\_DATE);  
System.out.println(formattedDate1); ***//2015-03-18***DateTimeFormatter germanFormatter = DateTimeFormatter.ofPattern("d. MMMM yyyy", Locale.GERMAN);  
String formattedDateGermany = germanFormatter.format(localDate);  
System.out.println(formattedDateGermany);***//18. März 2015***

We can parse the date from string also

DateTimeFormatter dateFormatter = DateTimeFormatter.ofPattern("yyyy-MM-dd");  
DateTimeFormatter timeFormatter = DateTimeFormatter.ofPattern("HH:mm:ss");  
DateTimeFormatter dateTimeFormatter = DateTimeFormatter.ofPattern("dd/MM/yyyy HH:mm:ss");  
  
String dateString = "2015-03-18";  
String timeString = "15:30:00";  
String dateTimeString = "18/03/2015 15:30:00";  
  
LocalDate localDate = LocalDate.parse(dateString,dateFormatter);  
LocalTime localTime = LocalTime.parse(timeString,timeFormatter);  
LocalDateTime localDateTime = LocalDateTime.parse(dateTimeString,dateTimeFormatter);

# Files

Different operating systems use different file systems to manage their data. But the JVM will automatically connect to the local file system, allowing you to perform the same operations across multiple platforms. Different operating systems vary in their format of pathnames. For example, Unix‐based systems use the forward slash, /, for paths, whereas Windows‐based systems use the backslash \ character. Java offers two options to retrieve the local separator character: a system property and a static variable defined in the File class

System.out.println(System.getProperty("file.separator"));

System.out.println(java.io.File.separator);

java.io.File class is used to read information about

● existing files and directories,

● list the contents of a directory,

● create/delete files and directories.

The File class cannot read or write data within a file, although it can be passed as a reference to many stream classes to read or write data

File operator usually takes a String which can be either an absolute path or relative path to the current directory. There are 3 ways to create files.

File file1 = new File("text.txt"); Pass the relative or absolute path

File file2 = new File("test.txt", "new.txt"); Pass the path of a parent file to create child file

File file3 = new File(file1, "new.txt"); Pass the reference of parent file

Note: Creating file object doesn't create a file in system

System.out.println(file1.exists()); //Without file will be false

## FileMethods

file1.exists(); Returns true if file/directory exists

file1.delete(); Returns true if file/directory deleted

file1.getAbsolutePath(); Returns String path of file/directory

file1.getName(); Returns string Name of file/directory

file1.getParent(); Returns parent folder directory returns null if passed like this "test.txt"

file1.isDirectory(); Returns true if it is directory

file1.isFile(); Returns true if it is file

file1.lastModified(); Returns long timeStamp

file1.length(); Return long file size

file1.listFiles(); Returns list of files in directory

file1.mkdir();

file1.mkdirs();

File dest = new File("new.txt");

System.out.println(file1.renameTo(dest));

It's important to notice from where we are running the java command in case of relative paths. For example if my class is in folder src-->javatest-->Test.java. It also has a file test.txt. Now if I run file.exists() from src as java .\javaTest\Test.java it will return false as it will check for test.txt in src. But if we run it from javatest then it will find the file and return true

## I/O Stream

The contents of a file may be accessed or written via a stream, which is a list of data elements presented sequentially. Streams are like long, nearly never‐ending “streams of water” with data presented one “wave” at a time. The stream is so large that once we start reading it, we have no idea where the beginning or the end is. We just have a pointer to our current position in the stream and read data one block at a time.

Each type of stream segments data into a “wave” or “block” in a particular way. For example, some stream classes read or write data as

* Individual bytes.
* Individual characters or strings of characters
* Larger groups of bytes or characters at a time.

Irrespective of how the data is read or written nearly all are built on top of reading or writing an individual byte or an array of bytes at a time.

The reason higher-level streams exist is for convenience, as well as performance. For example, writing a file one byte at a time is time-consuming and slow in practice because the round-trip between the Java application and the file system is relatively expensive. By utilizing a BufferedOutputStream, the Java application can write a large chunk of bytes at a time, reducing the round-trips and drastically improving performance.



Files can be big like 1 TB. What stream does is helps reading chunks of data thus allowing systems with smaller specifications to be able to read data.

## Stream Types

### Byte vs. Character Streams

Stream can be classified as

|  |  |
| --- | --- |
| Byte Stream | Character streams |
| Which read/write binary data ( 0s and 1s) | Which read/write text data |
| have class names that end in InputStream or OutputStream | have class names that end in Reader or Writer |
| The byte streams are primarily used to work with binary data, such as an image or executable file | character streams are used to work with text files |
| FileInputStream | FileReader |

Even though character streams do not contain the word Stream in their class name, they are still I/O streams

Byte stream classes can write all types of binary data, including strings. So one might think that Character streams like reader are not necessary. However Character streams specifically focused on managing character and string data so that one doesn’t worry about the underlying character encoding of the file.

Java supports a wide variety of character encodings

Charset usAsciiCharset = Charset.forName("US-ASCII");

Charset utf8Charset = Charset.forName("UTF-8");

Charset utf16Charset = Charset.forName("UTF-16");

### Input VS Output Streams

Most InputStream stream classes have a corresponding OutputStream class, and vice versa. For example, the FileOutputStream class writes data that can be read by a FileInputStream. If you understand the features of a particular Input or Output stream class, you should naturally know what its complementary class does

It follows, then, that most Reader classes have a corresponding Writer class. For example, the FileWriter class writes data that can be read by a FileReader

Exception

**PrintWriter** no reader **PrintStream** has no Input stream

### Low vs High level Stream

A low‐level stream connects directly with the source of the data, such as a file, an array, or a String. Low‐level streams process the raw data or resource and are accessed in a direct and unfiltered manner

A high‐level stream is built on top of another stream using wrapping. Wrapping is the process by which an instance is passed to the constructor of another class

        try (var reader = new BufferedReader(new FileReader("text.txt"))) {

            System.out.println(reader.readLine());

        } catch (IOException e) {

            e.printStackTrace();

        }

In this example, FileReader is the low‐level stream reader, whereas BufferedReader is the high‐level stream that takes a FileReader as input. Many operations on the high‐level stream pass through as operations to the underlying low‐level stream, such as read() or close(). Other operations override or add new functionality to the low‐level stream methods. The high‐level stream may add new methods, such as readLine(), as well as performance enhancements for reading and filtering the low‐level data.

High‐level streams can take other high‐level streams as input

        try (var ois = new ObjectInputStream(new BufferedInputStream(new FileInputStream("text.txt")))) {

            System.out.print(ois.readObject());

        } catch (ClassNotFoundException e) {

            e.printStackTrace();

        } catch (IOException e) {

            e.printStackTrace();

        }

Buffered classes read or write data in groups, rather than a single byte or character at a time. The performance gain from using a Buffered class to access a low‐level file stream cannot be overstated. Unless you are doing something very specialized in your application, you should always wrap a file stream with a Buffered class in practice.

The java.io library defines four abstract classes that are the parents of all stream classes defined within the API: **InputStream, OutputStream, Reader, and Writer**. The constructors of high‐level streams often take a reference to the abstract class. For example, BufferedWriter takes a Writer object as input, which allows it to take any subclass of Writer.

        InputStream (Abstract) --> FileInputStream --> BufferedInputStream

        OutputStream(Abstract) --> FileOutputStream--> BufferedOutputStream

        Reader(Abstract)--> FileReader --> BufferedReader

        Writer(Abstract)-->FileWriter -->BufferedWriter

        new BufferedInputStream(new FileReader("z.txt")); *// DOES NOT COMPILE Reader wrapped in Input Stream*

        new BufferedWriter(new FileOutputStream("z.txt")); *// DOES NOT COMPILE Same*

        new ObjectInputStream( new FileOutputStream("z.txt")); *// DOES NOT COMPILE output wrapped in input*

### Summary

|  |  |  |
| --- | --- | --- |
| Class Name | Type | Description |
| InputStream | Abstract | Abstract class for all input byte streams |
| OutputStream | Abstract | Abstract class for all output byte streams |
| Reader | Abstract | Abstract class for all input character streams |
| Writer | Abstract | Abstract class for all output character streams |
| FileInputStream | Low | Reads file data as bytes |
| FileOutputStream | Low | Writes file data as bytes |
| FileReader | Low | Read file data as character |
| FileWriter | Low | Write file data as character |
| BufferedInputStream | High | Reads byte data from an existing InputStream in a buffered manner, which improves efficiency and performance |
| BufferedOutputStream | High | Writes byte data from an existing OutputStream in a buffered manner, which improves efficiency and performance |
| BufferedReader | High | Reads character data from an existing Reader in a buffered manner, which improves efficiency and performance |
| BufferedWriter | High | Writes character data to an existing Writer in a buffered manner, which  improves efficiency and performance |
| ObjectInputStream | High | Deserializes primitive Java data types and graphs of Java objects from an existing InputStream |
| ObjectOutputStream | HIgh | Serializes primitive Java data types and graphs of Java objects to an existing OutputStream |
| PrintStream | High | Writes formatted representations of Java objects to a binary stream |
| PrintWriter | High | Writes formatted representations of Java objects to a character stream |

## Using FileInputStream and FileOutputStream

        File fileIn = new File("in.txt");

        File fileOut = new File("out.txt");

        try (var in = new FileInputStream(fileIn);

                var out = new FileOutputStream(fileOut)) {

            int data;

            while ((data = in.read()) != -1) {

                out.write(data);

            }

        } catch (IOException e) {

            e.printStackTrace();

        }

The FileOutputStream class includes overloaded constructors that take a boolean append flag. When set to true, the output stream will append to the end of a file if it already exists

var out = new FileOutputStream(fileOut, true);

# Functional Programming

From Java 8 a new style of programming is introduced. This style we call it functional programming which we can build by using the lambda expressions. Before Java 8, all the developers they used to write the code in an imperative style.  from Java 8 the developers they have an option to develop their programs either using the old traditional imperative style or with the new style of programming, which is functional programming. Imperative approach is available inside Java from day one. It has its own fair share of accidental complexities, whereas with the functional programming

* We should be able to write or develop complex business logic with the few lines of code.
* Whatever code that we are going to build with the help of functional programming style, it is going to enhance the readability of our code since the code which is developed with the help of functional programming is going to more look like a business requirement written in a plain English.

## Lambda

A lambda expression/function is a short block of code which takes in parameters and can return a value or execute some statements. Lambda expressions are like methods, but they do not need a name, and they can be implemented right in the body of a method.

Syntax of lambda expression is

(parameters) 🡪 {statements}

if the lambda function has one statement, we can write like this

(parameter) -> statement;

Rules of Lambda

* We need to pass parameters in () followed by -> and our statements we can put inside {}. If we have single statement {} is optional

|  |  |
| --- | --- |
| *public void* test() {  System.out.println("Hello");  System.out.println("World"); } | () -> {  System.out.println("Hello");  System.out.println("World"); }; |
| *public void* test() {  System.out.println("Hello"); } | () -> System.out.println("Hello"); |

* For passing multiple params to Lambda () is to be used, it is not needed for single params

s -> System.out.println(s).

(s, r) -> System.out.println(s);

|  |  |
| --- | --- |
| *public void* add(*int* i, *int* j) {  System.out.println(i+j); } | (i, j) -> System.out.println(i +j); |
| *public void* doubleValue(*int* i) {  System.out.println(i\*2); } | i -> System.out.println(i \*2); |

* return keyword: return keyword is implied and is optional **s -> {return true;};** is same as  **s -> true;** However for multiple line statements or if we use {}return keyword is must

|  |  |
| --- | --- |
| public int add(int i, int j) {  return (i+j); } | (i, j) -> {  *return* (i + j); };  Or  (i, j) -> i + j; |

* It is a good practice to keep one or two lines of code in Lambda expression (not a rule but more of a best practice).

For a Lambda we need a functional interface. **Functional Interface** in Java is an interface that contains only one abstract method. It can have any number of default or static methods but must contain exactly one abstract method. We can annotate with @*FunctionalInterface* which will force the interface to have single abstract method. This will prevent anyone else from making it not a functional interface by adding another abstract method.

@FunctionalInterface  
interface DoSomething {  
  
 void doSomething();  
}

The way we use lambda expression is that we can override the functional interface in the Lambda code itself without the need of creating an implementation of Functional interface or writing an inner class implementation. For e.g. here we have a FI with abstract method test. We created an inline implementation and called the test method.

*public class* Test {  
  
 *public static void* main(String[] args) {  
 DoSomething doSomething = () -> System.out.println("doing");  
 doSomething.test();  
 }  
}  
  
  
*@FunctionalInterface  
interface* DoSomething {  
  
 *void* test();  
}

A Lambda expression always needs a functional interface method to execute i.e. interface with one abstract method. It doesn't work with Abstract class.

With lambda expression we can pass the implementation as a reference to method. In imperative programming we can pass only data in form method parameters like objects and parameters, but functional programming allows us to pass implementation as well

*public class* Test {  
  
 *public static void* main(String[] args) {  
 DoSomething doSomething = () -> System.out.println("doing");  
 DoSomething doSomething1 = () -> System.out.println("doing");  
 method(doSomething); ***// passing implementation as parameter*** method(doSomething1); ***// passing implementation as parameter*** }  
  
 *public static void* method(DoSomething doSomething) {  
 doSomething.test();  
 }  
}  
  
*@FunctionalInterface  
interface* DoSomething {  
  
 *void* test();  
}

Some other examples, here we can see that with same FI we can pass different implementation as per our requirements

*public class* Test {  
  
 *public static void* main(String[] args) {  
 Calculate calculate = (i, j) -> i+ j;  
 System.out.println(operation(calculate, 2,3)); ***// Passing the implementation to method*** Calculate calculate1 = ((i, j) -> i-j);  
 System.out.println(calculate1.calculate(4,2)); ***// Directly calling FI*** Calculate calculate2 = ((i, j) -> i\*j);  
 System.out.println(calculate2.calculate(4,2));  
  
 Calculate calculate3 = ((i, j) -> i/j);  
 System.out.println(calculate3.calculate(4,2));  
  
 }  
  
 *public static int* operation(Calculate calculate, *int* i, *int* j) {  
 *return* calculate.calculate(i, j);  
 }  
  
}  
  
*@FunctionalInterface  
interface* Calculate {  
 *int* calculate(*int* i, *int* j);  
}

Usual way of working with interface is that

1. We must create a class implementing it, provide the implementation and initialize it and then calling the method. But with this approach lines of code increases, and application becomes complicated, and they create extra class files which can bloat the footprint of application

|  |  |
| --- | --- |
| public class Test {  public static void main(String[] args) {  ShoutImpl sh = new ShoutImpl();  sh.shout();  }  }  @FunctionalInterface  interface Shout {  void shout();  }  class ShoutImpl implements Shout {  @Override  public void shout() {  System.out.println("Interface");  }  } |  |

1. Or create an inner class implementation. Here we create an anonymous inner class implementation, but this also makes the application complicated and bloated. Also, since this doesn’t mandate the use of Functional interface so if the interface have more than one abstract method we have to implement all the abstract methods weather we need it or not.

|  |  |
| --- | --- |
| public class Test {  public static void main(String[] args) {  Shout shoutInnerClass = new Shout() {  public void shout() {  System.out.println("Inner Class");  }  };  shoutInnerClass.shout();  }  }  @FunctionalInterface  interface Shout {  void shout();  } | A screenshot of a computer  Description automatically generated |

But with Lambda we can provide the implementation in line. And we don’t see any extra class file which can bloat our application size.

|  |  |
| --- | --- |
| public class Test {  public static void main(String[] args) {  Shout sh = () -> System.out.println("Lambda");  sh.shout();  }  }  @FunctionalInterface  interface Shout {  void shout();  } |  |

## Functional Interface

With lambda function we still see one drawback that we must create FI ourselves. This seems unnecessary as it increase the footprint of our application. Hence java provides some common FI predefined in JDK. We can utilize these and hence eradicate the necessity of writing FI which seems like boilerplate code. The out of the box Functional Interface provided by java will cover most of the scenarios needed by developers. For some rare cases we can still create FI, but mostly it’s not needed. List of predefined FI are mentioned below.

|  |  |  |  |
| --- | --- | --- | --- |
| FI | method | info | Usage |
| Predicate<T> | test(T) | Used to test condition return boolean | Predicate<String> p1 = String::isEmpty;  Predicate<String> p2 = x -> x.isEmpty();  System.out.println(p1.test(""));  System.out.println(p2.test("")); |
| Function<T,R> | apply(T) | Take input give result | Function<String, Integer> f1 = String::length;  Function<String, Integer> f2 = x -> x.length();  System.out.println(f1.apply("cluck"));  System.out.println(f2.apply("cluck")); |
| UnaryOperator<T> | apply(T,U) | Input and output parameters type is same, similar to Functions use in case both parameters are same | UnaryOperator<String> u1 = String::toUpperCase;  UnaryOperator<String> u2 = x -> x.toUpperCase();  System.out.println(u1.apply("chirp"));  System.out.println(u2.apply("chirp")); |
| Consumer<T> | accept(T) | do something with a parameter but not return anything | Consumer<String> c1 = System.out::println;  Consumer<String> c2 = x -> System.out.println(x);  c1.accept("Annie");  c2.accept("Annie"); |
| Supplier<T> | get() | generate or supply values without taking any input, like creating objects. | Supplier<LocalDate> s1 = () -> LocalDate.now();  Supplier<LocalDate> s2 = LocalDate::now;  System.out.println(s1.get()+ s1.get()); |
| BiPredicate<T,U> | test(T,U) | Same as Predicate takes 2 input | BiPredicate<String, String> b1 = String::startsWith;  BiPredicate<String, String> b2 = (s, p) -> s.startsWith(p);  System.out.println(b1.test("chicken", "chick"));  System.out.println(b2.test("chicken", "chick")); |
| BiFunction<U,T,R> | apply(U,T) | Same as function accepts 2 input | BiFunction<String, String, String> b1 = String::concat;  BiFunction<String, String, String> b2 = (s, b) -> s.concat(b);  System.out.println(b1.apply("baby ", "chick"));  System.out.println(b2.apply("baby ", "chick")); |
| BiConsumer<T,U> | accept(T,U) | Same as consumer accepts 2 values | var map = new HashMap<String, Integer>();  BiConsumer<String, Integer> b1 = map::put;  BiConsumer<String, Integer> b2 = (k, v) -> map.put(k, v);  b1.accept("chicken", 7);  b2.accept("chick", 1); |
| BinaryOperator<T> | apply(T,U) | Replacement of BiFunction when all input and return is of same type | BinaryOperator<String> b1 = String::concat;  BinaryOperator<String> b2 = (s, b) -> s.concat(b);  System.out.println(b1.apply("baby ", "chick"));  System.out.println(b2.apply("baby ", "chick")); |

**1Predicate<T> :** Takes an input return boolean. It can be used to test any condition hence it has an abstract method boolean test(T) . It also has static methods to chain the implementation with other predicates. and(), or(), not(), negate(), isEqual()

Predicate<Integer> isEven = num -> num%2==0;  
System.out.println(isEven.test(16)); ***// true***System.out.println(isEven.test(13)); ***// false***Predicate<Integer> isGreaterThan50 = num -> num>50;  
System.out.println(isGreaterThan50.test(61)); ***// true***System.out.println(isGreaterThan50.test(31)); ***// false  
  
// To check if the number is even and greater than 50***System.out.println(isEven.and(isGreaterThan50).test(63)); ***// false logical and operation  
  
// To check if the number is even or greater than 50***System.out.println(isEven.or(isGreaterThan50).test(63)); ***// true logical or operation***System.out.println(isEven.negate().test(63)); ***// true logical negate operation***Predicate<Integer> isOdd = Predicate.not(isEven);  
System.out.println(isOdd.test(4)); ***// false***Predicate<String> checkEquality = Predicate.isEqual("Eazy Bytes");  
System.out.println(checkEquality.test("Eazy Bytes")); ***// true***

2 **Function<T,R> :** Represents a function that accepts one argument T and produces a result R. Has abstract method R apply(T). It can take an input of one type and return something different.

Predicate<Integer> isEven = num -> num%2==0;  
System.out.println(isEven.test(16)); // true  
System.out.println(isEven.test(13)); // false  
  
Predicate<Integer> isGreaterThan50 = num -> num>50;  
Function<String, String> convertStr = input -> input.toUpperCase();  
System.out.println(convertStr.apply("Eazy Bytes"));  
  
Function<String, Integer> getStrLength = input -> input.length();  
System.out.println(getStrLength.apply("Eazy Bytes"));  
  
Function<String, String> sameValue = Function.identity(); // returns same value  
System.out.println(sameValue.apply("Hi Madan"));  
  
Function<Integer, Integer> doubleValue = num -> num \* 2;  
Function<Integer, Integer> addThree = num -> num + 3;  
  
Function<Integer, Integer> output1 = doubleValue.andThen(addThree); // first left side will execute then right side  
Function<Integer, Integer> output2 = doubleValue.compose(addThree); // first right side will execute then left side  
System.out.println(output1.apply(5)); // 13  
System.out.println(output2.apply(5)); // 16

3 **UnaryOperator<T> :** Represents a function that accepts an argument and produces a result of the same type. Has abstract method T apply(T). This extends Function but the difference is that the return type and arguments are of the same type as compared to Function. Its convenient to use when both input and out put are of same type

UnaryOperator<String> convertStr = input -> input.toUpperCase();  
System.out.println(convertStr.apply("Eazy Bytes"));  
  
UnaryOperator<String> sameValue = UnaryOperator.identity();  
System.out.println(sameValue.apply("Hi Madan"));  
  
UnaryOperator<Integer> doubleValue = num -> num \* 2;  
UnaryOperator<Integer> addThree = num -> num + 3;  
  
Function<Integer, Integer> output1 = doubleValue.andThen(addThree);  
Function<Integer, Integer> output2 = doubleValue.compose(addThree);  
System.out.println(output1.apply(5)); // 13  
System.out.println(output2.apply(5)); // 16

4 **Consumer<T> :** Returns nothing and has 1 input. Has abstract method accept(T), You use a Consumer when you want to do something with a parameter but not return anything

Consumer<String> convertAndDisplay = input -> System.out.println(input.toUpperCase());  
convertAndDisplay.accept("Eazy Bytes");  
  
Consumer<Integer> squareOf = num -> System.out.println(num \* num);  
List<Integer> numbersList = Arrays.asList(1,2,3,4,5,6,7,8,9,10);  
numbersList.forEach(squareOf);  
  
Consumer<String> appendInput = input -> System.out.println("New value after appending is : HELLO " + input);  
appendInput.andThen(convertAndDisplay).accept("Lambda Expression"); // One lambda will be executed after another

5 **Supplier<T>** : Returns an object and has no input. Has abstract method <T> get(). A Supplier is used when you want to generate or supply values without taking any input, like creating objects.

Supplier<Integer> getCurrentMonth = () -> LocalDate.now().getMonthValue();  
System.out.println(getCurrentMonth.get());  
  
Supplier<Integer> getCurrentDayOfMonth = () -> LocalDate.now().getDayOfMonth();  
System.out.println(getCurrentDayOfMonth.get());  
  
Supplier<String> getCurrentDayName = () -> LocalDate.now().getDayOfWeek().name();  
System.out.println(getCurrentDayName.get());

6 **BiPredicate<T,U>** : Takes 2 inputs and returns a boolean. Has an abstract method boolean test(T,U) .

BiPredicate<String, String> b1 = String::startsWith;

BiPredicate<String, String> b2 = (string, prefix) -> string.startsWith(prefix);

System.out.println(b1.test("chicken", "chick"));

System.out.println(b2.test("chicken", "chick"));

BiPredicate<Integer, Integer> isSumEven = (num1, num2) -> (num1+num2) % 2 == 0;  
System.out.println(isSumEven.test(4,9)); // false

7 **BiFunction<U,T,R> :** Represents a function that accepts two arguments and produces a result. Has abstract method R apply(U,T) .

BiFunction<String, String, String> b1 = String::concat;

BiFunction<String, String, String> b2 = (string, toAdd) -> string.concat(toAdd);

System.out.println(b1.apply("baby ", "chick"));

System.out.println(b2.apply("baby ", "chick"));

BiFunction<Double,Double,Double> calculatePower = (num1, num2) -> Math.pow(num1, num2);  
System.out.println(calculatePower.apply(2.0,3.0));

8 **BiConsumer<T,U>** : Returns nothing and has 2 input. Has abstract method accept(T,U)

var map = new HashMap<String, Integer>();

BiConsumer<String, Integer> b1 = map::put;

BiConsumer<String, Integer> b2 = (k, v) -> map.put(k, v);

b1.accept("chicken", 7);

b2.accept("chick", 1);

System.out.println(map);

9 **BinaryOperator<T> :** It is a counter part of Bi function where all input and output are of same type. Just like Function has a Urnary operator similarly BiFunction has BinaryOperator where input and output type is same Has abstract method T apply(T) .

BinaryOperator<String> b1 = String::concat;

BinaryOperator<String> b2 = (firstString, secondString) -> firstString.concat(secondString);

System.out.println(b1.apply("baby ", "chick"));

System.out.println(b2.apply("baby ", "chick"));

Scope of Lambda Expression

Lambda expression is not going to create its own scope. Instead, it is going to work within the scope of the enclosing context. This characteristic is referred as lexical scoping inside lambda expressions concept. As a result, below code will not compile as input is already defined outside Lambda

String input = "Hello World";  
Consumer<String> consumer = input -> { //This will not compile name of input needs to be changed  
 System.out.println(input+input);  
};  
consumer.accept(input);

But below will compile

Consumer<String> consumer = input -> {  
 System.out.println(input);  
};  
String input = "Hello World"; // This will compile as variable input in Lambda is not available here  
consumer.accept(input);

Below will not compile. Because Variable used in lambda expression should be final or effectively final. This is enforced because inside the multithreading environment, we will get unexpected results, because each execution of your lambda expression is trying to have different,

different values.

String input = "Hello World";  
Consumer<String> consumer = input1 -> {  
 input = "Hi World"; //This will not compile  
 System.out.println(input1+input);  
};  
consumer.accept(input);

Even if we change the value outside lambda it will not compile like below

String input = "Hello World";  
input = "Hi World";   
Consumer<String> consumer = input1 -> {  
 System.out.println(input1+input); //This will not compile  
};  
consumer.accept(input);

Can create and modify variables inside Lambda

String input = "Hello World";  
Consumer<String> consumer = input1 -> {  
 String msg = "Hi";  
 msg = "wow"; // Will compile  
 System.out.println(input1+input + msg);  
};  
consumer.accept(input);

## Method Reference

Method reference is used to refer to a method of functional interface. It is a compact and easy form of lambda expression. When we simply pass the parameters of Lambda expression to functional interface implementation, we can instead use Method reference instead of Lambda which looks much cleaner. You can pretend the compiler turns your method references into lambdas for you.

|  |  |
| --- | --- |
| TestClass obj = (text) -> System.out.println(text);  obj.testMethod("Dummy Text"); | TestClass obj= System.out::println;  obj.testMethod("Dummy Text"); |

There are four formats for usage of method references, depending on where we are using it.

1. **CALLING STATIC METHODS :** Collection.sort(lst) becomes Collection::sort

Consumer<List<Integer>> methodRef = Collections::sort;

methodRef.accept(lst);

Consumer<List<Integer>> lambda = s-> Collections.sort(s);

lambda.accept(lst);

1. CALLING INSTANCE METHODS ON OBJECT AND PASSING A PARAM: str.startsWith("a") becomes str::startsWith("a")

var str = "abc";

Predicate<String> methodRef = str::startsWith;

methodRef.test("a");

var str = "abc";

Predicate<String> lambda = s -> str.startsWith(s);

lambda.test("a");

1. CALLING INSTANCE METHODS ON OBJECT : str.isEmpty()

Predicate<String> methodRef = String::isEmpty;

methodRef.test("Yo");

Predicate<String> lambda = s -> s.isEmpty();

lambda.test("Yo");

1. CALLING CONSTRUCTORS

Supplier <List<String>> methodRef4 = ArrayList::new;

methodRef4.get();

Supplier<List<String>> lambda4= () -> new ArrayList();

lambda.get();

## Optional

Java introduced a new class Optional in jdk8. It is a public final class and used to deal with NullPointerException in Java applications. It provides methods which are used to check the presence of value for a particular variable.

Suppose we have an average method which takes an array of integers and calculates average for all the marks scored by a student. Now suppose a student was sick and did not take the exam, so input array length will be zero and so will be the average. This is not correct as average should not be applicable in this scenario. This we can handle using optional

public static Optional<Double> average(int... scores) {

if (scores.length == 0)

return Optional.empty(); *// Sets empty optional*

double sum = 0;

for (int score : scores)

sum += score;

return Optional.of( sum / scores.length); *// Sets value using of Method*

}

If no scores are passed then the method will return an empty optional, otherwise it will return the average wrapped in optional.

System.out.println(average(90, 100)); *// Optional[95.0]*

System.out.println(average()); *// Optional.empty*

Instead of using of() and empty() separately we can use offNullable

    public static void main(String[] args) {

        System.out.println(getMessage(1));

        System.out.println(getMessage(2));

    }

    public static Optional<String> getMessage(int i){

        String message = null;

        if(i%2 ==0)

            message ="Hello";

        return Optional.ofNullable(message);

    }

As you can see above the Optional returns the result wrapped in Optional like Optional[95.0] instead of 95.0.So if we want to get the Double value then the Optional method provides a get() method to retrieve value from optional. But we must check if the value is present or not with ifPresent() method otherwise it will throw NoSuchElementException

Optional<Double> opt = average(90, 100);

if (opt.isPresent())

System.out.println(opt.get()); *// 95.0*

opt = average();

System.out.println(opt.get()); *// java.util.NoSuchElementException*

ifPresent() method does the same except that it Calls Consumer with value

opt.ifPresent(System.out::println); *//95.0*

Other useful methods

Setting data from optional

* Optional.empty() // Sets null value calling get will cause
* Optional.of(9) // Sets value
* Optional.ofNullable(null) // can be used to set value

Getting data

Optional<Double> opt2 = average();

* System.out.println(opt2.get()); //Gets the value, will be throws null pointer if value is null
* System.out.println(opt2.orElse(Double.NaN));
* System.out.println(opt2.orElseGet(() -> Math.random()));
* System.out.println(opt2.orElseThrow()); *// NoSuchElementException*
* System.out.println(opt2.orElseThrow(() -> new IllegalStateException())); *// Throws exception created by calling Supplier*

## Stream

A Stream in Java represents a sequence of elements and supports different operations to perform computations on these elements. Streams allow for functional-style operations to be performed on collections of objects, such as filtering, mapping, reducing, and more. Stream provides a feature called stream pipeline which consists of the operations that run on a stream to produce a result. There are three parts to a stream pipeline

* **Source**: Where the stream is generated
* **Intermediate operations**: Can transforms in stream. There can be one or more intermediate operations. Since streams use lazy evaluation, the intermediate operations do not run until the terminal operation runs.
* **Terminal operation**: Actually, produces a result from stream. Since streams can be used only once, the stream is no longer valid after a terminal operation completes.



Creating Stream or source

Empty Stream

Stream<String> empty = Stream.empty();

Stream from elements

Stream<Integer> singleElement = Stream.of(1);

Stream<Integer> fromArray = Stream.of(1, 2, 3);

Stream from collections

var list = List.of("a", "b", "c");

Stream<String> fromList = list.stream();

Stream class provides generate method which takes a Supplier as input to create infinite stream

Random rd = new Random();

Stream<Integer> intStream = Stream.generate(rd::nextInt);

intStream.limit(5).forEach(System.out::println);

Infinite stream which generates Random Double with Supplier

Stream<Double> doubleStream = Stream.generate(Math::random);

doubleStream.limit(5).forEach(System.out::println);

Infinite Stream of String "YO"

Stream<String> stringStream = Stream.generate(() -> "YO");

stringStream.limit(5).forEach(System.out::println);

Stream also provides an iterate method to create streams. It takes in a seed which is the initial value and an unary operator to generate next values. This generates an infinite stream.

var intStream = Stream.iterate(1, s -> s + 1);

intStream.limit(5).forEach(System.out::println);

There is another overloaded version of iterate which takes a seed, an unary operator and a predicate to determine condition, This was introduced in Java 9 and the predicate can be used to make the stream finite. When the predicate returns false the stream stops

var intStream = Stream.iterate(0, s -> s < 20, s -> s + 1);

intStream.forEach(System.out::println);

Features

* **Lazy Evaluation**: Streams perform operations only when necessary. Intermediate operations are typically lazy, meaning they do not compute a result until the result is needed by a terminal operation.
* **Parallel Processing**: Streams can leverage parallel processing, making it easier to write parallelizable code to take advantage of multi-core processors.
* **Functional Composition**: Streams provide methods for composing operations in a functional style, making it easy to chain multiple operations together.
* **No Storage**: Streams in Java are not data storage containers; instead, they provide a way to process elements from a source (such as a collection, array, or I/O channel) in a functional and declarative style. Streams operate on data elements in a sequence and allow for various operations to be performed on those elements, such as filtering, mapping, sorting, and reducing.

## Stream Terminal Operations

1 **count() :** The count() method determines the number of elements in a finite stream. For an infinite stream, it never terminates

Stream<String> s = Stream.of("monkey", "gorilla", "bonobo");

System.out.println(s.count());

2 **min() and max() :** The min() and max() methods allow you to pass a custom comparator and find the smallest or largest value in a finite stream according to that sort order. Like the count() method, min() and max() hang on an infinite stream. It returns a false for empty stream

Optional<T> min(Comparator<? super T> comparator)

Optional<T> max(Comparator<? super T> comparator)

Stream<String> s = Stream.of("monkey", "donkey", "bonobo");

Optional<String> min = s.min((s1, s2) -> s1.length()-s2.length());

min.ifPresent(System.out::println);

3 **findAny() and findFirst() :** The findAny() and findFirst() methods return an element of the stream unless the stream is empty. If the stream is empty, they return an empty Optional. It can terminate an infinite stream

Stream<Double> randoms = Stream.generate(Math::random);

randoms.findAny().ifPresent(System.out::println);

Stream<Integer> oddNumbers = Stream.iterate(1, n -> n + 2);

oddNumbers.findFirst().ifPresent(System.out::println);

4 **allMatch(), anyMatch(), and noneMatch() :** The findAny() and findFirst() methods return an element of the stream unless the stream is empty. If the stream is empty, they return an empty Optional. It can terminate an infinite stream

anyMatch allMatch --> boolean = stream.anyMatch(Predicate <? super T> predicate)

var list = List.of("monkey", "2", "chimp");

Predicate<String> pred = x -> Character.isLetter(x.charAt(0));

System.out.println(list.stream().anyMatch(pred)); *// true*

System.out.println(list.stream().allMatch(pred)); *// false*

System.out.println(list.stream().noneMatch(pred)); *// false*

5 **forEach() :** Iterates over each element in a Stream, Won't work for infinite stream

Stream<String> s1 = Stream.of("Monkey", "Gorilla", "Bonobo");

s1.forEach(System.out::print);

6 **reduce() :** The reduce() method combines a stream into a single object. It is a reduction, which means it processes all elements. Reduction stream operations allow us to produce one single result from a sequence of elements, by repeatedly applying a combining operation to the elements in the sequence. It has 3 signatures.

REDUCE example 1 ->> T reduce(T identity, BinaryOperator<T> accumulator)

Stream<String> stream = Stream.of("w", "o", "l", "f");

Stream<String> newstream = Stream.of("w", "o", "l", "f");

String word = stream.reduce("", (x, c) -> x + c);

String word2 = newstream.reduce("", String::concat);

System.out.println(word);

System.out.println(word2);

REDUCE example 2 -->> **Optional<T> reduce(BinaryOperator<T> accumulator)** When identity is not provided it returns an optional

BinaryOperator<Integer> op = (a, b) -> a \* b;

Stream<Integer> empty = Stream.empty();

empty.reduce(op).ifPresent(System.out::println); *// no output*

Stream<Integer> oneElement = Stream.of(3);

oneElement.reduce(op).ifPresent(System.out::println); *// 3*

Stream<Integer> threeElements = Stream.of(3, 5, 6);

threeElements.reduce(op).ifPresent(System.out::println); *// 90*

REDUCE example 3 -->> <U> U reduce(U identity, BiFunction<U,? super T,U> accumulator, BinaryOperator<U> combiner) The third method signature is used when we are dealing with different types. It allows Java to create intermediate reductions and then combine them at the end

Stream<String> stream1 = Stream.of("w", "o", "l", "f!");

int length = stream1.reduce(0,

(i, w) -> i + w.length(),

(a, b) -> a + b);

System.out.println(length);

7 **collect**() : The collect() function is used to accumulate the elements of a stream into a collection or a single value It lets us get data out of streams and into another form

<R,A> R collect(Collector<? super T,A,R> collector);

        List<String> departmentList = new ArrayList<>();

        departmentList.add("Supply");

        departmentList.add("HR");

        departmentList.add("Sales");

        departmentList.add("Marketing");

        Set<String> newDepartmentList = departmentList.stream()

                .filter(word -> word.startsWith("S"))

                .collect(Collectors.toCollection(TreeSet::new));

        newDepartmentList.forEach(System.out::println);

        long totalCount = departmentList.stream()

                .filter(word -> word.startsWith("S"))

                .collect(Collectors.counting());

        System.out.println(totalCount);

8 **collectingAndThen** ()The collectingAndThen() function is used to perform an additional operation on the result of the collect() function.

<R,A> R collectingAndThen(Collector<? super T,A,R> downstream, Function<R,RR> finisher);

* **downstream :**A Collector that describes how to accumulate elements into a result container.
* **finisher :**The function to be applied to the result of the collection operation.

class Product {

    String name;

    Integer price;

    public Product(String name, Integer price) {

        this.name = name;

        this.price = price;

    }

}

public class Test {

    public static void main(String[] args) {

        List<Product> productList = Arrays.asList(new Product("Apple", 1200),

                new Product("Samsung", 1000), new Product("Nokia", 600),

                new Product("BlackBerry", 1000), new Product("Apple Pro Max", 1500),

                new Product("Mi", 800), new Product("OnePlus", 1000));

        String maxPriceProduct = productList.stream().collect(Collectors.collectingAndThen(

                Collectors.maxBy(Comparator.comparing(product -> product.price)),

                (productOptional -> productOptional.isPresent() ? productOptional.get().name : "None")));

        System.out.println(maxPriceProduct);

    }

}

9 **groupingBy()** The groupingBy() function in Java Streams is used to group elements of a stream by a classification function.

groupingBy(Function<? super T, ? extends K> classifier);

The classification function used to group elements.

        List<String> words = Arrays.asList("apple", "banana", "cherry", "blueberry", "avocado");

        Map<Character, List<String>> groupedByFirstLetter = words.stream()

                .collect(Collectors.groupingBy(word -> word.charAt(0)));

        System.out.println("Grouped by first letter: " + groupedByFirstLetter);

        //Grouped by first letter: {a=[apple, avocado], b=[banana, blueberry], c=[cherry]}

10 partitioningBy() Function

The partitioningBy() function in Java Streams is used to partition elements into two groups based on a predicate.

        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        Map<Boolean, List<Integer>> partitionedEvenOdd = numbers.stream()

                .collect(Collectors.partitioningBy(num -> num % 2 == 0));

        System.out.println("Partitioned into even and odd: " + partitionedEvenOdd);

        //Partitioned into even and odd: {false=[1, 3, 5, 7, 9], true=[2, 4, 6, 8, 10]}

## Stream Intermediate Operations

Unlike a terminal operation, an intermediate operation produces a stream as its result. An intermediate operation can also deal with an infinite stream simply by returning another infinite stream

1 **filter() :** The filter() method returns a Stream with elements that match a given expression. Here is the method

signature: Stream<T> filter(Predicate<? super T> predicate)

Stream<String> stream = Stream.of("monkey", "gorilla", "bonobo");

stream.filter(x -> x.startsWith("m")).forEach(System.out::print);*// monkey*

2 **distinct() :** The distinct() method returns a stream with duplicate values removed. Java calls equals() to determine whether the objects are the same

Stream<String> s = Stream.of("duck-", "duck-", "duck-", "goose");

s.distinct().forEach(System.out::print);*//duck-goose*

3 **limit() and skip() :** The limit() and skip() methods can make a Stream smaller, or they could make a finite stream out of an infinite stream

Stream<Integer> s2 = Stream.iterate(1, n -> n + 1);

s2.skip(5).limit(2).forEach(System.out::print);

4 **map() :** The  map() is used to transform one object into another by applying a business logic with a lambda expression.

<R> Stream<R> map(Function<? super T, ? extends R> mapper)

Stream<String> s3 = Stream.of("monkey", "gorilla", "bonobo");

s3.map(String::length).forEach(System.out::print);

5 **flatMap() :** The flatMap() function allows flattening of nested collections or streams within a stream, resulting in a single stream of elements. So when each element of Stream is it self a stream or collection we need to flatten the element then we can extract individual item

List<String> zero = List.of();

var one = List.of("Bonobo");

var two = List.of("Mama Gorilla", "Baby Gorilla");

Stream<List<String>> animals = Stream.of(zero, one, two);

animals.flatMap(m -> m.stream()).forEach(System.out::println);

        String[] arrayOfWords = { "Eazy", "Bytes" };

        Stream<String> streamOfWords = Arrays.stream(arrayOfWords);

        Stream<String[]> streamOfLetters = streamOfWords.map(word -> word.split(""));

        streamOfLetters.flatMap(Arrays::stream).forEach(System.out::println);

6 **sorted() :** The sorted() method returns a stream with the elements sorted. Java uses natural ordering unless we specify a comparator.

Stream<String> stream2 = Stream.of("brown-", "bear-");

stream2.sorted().forEach(System.out::print); *// bear-brown*

Stream<String> stream3 = Stream.of("brown bear-", "grizzly-");

stream3.sorted(Comparator.reverseOrder()).forEach(System.out::print); *// grizzly-brown bear*

Take a look at the method signatures again. Comparator is a functional interface. This means that we can use method references or lambdas to implement it. The Comparator interface implements one method that takes two String parameters and returns an int. However, Comparator::reverseOrder doesn't do that. It is a reference to a function that takes zero parameters and returns a Comparator. This is not compatible with the interface. This means that we have to use a method and not a method reference

stream3.sorted(Comparator::reverseOrder);

7 **peek() :** It is useful for debugging because it allows us to perform a stream operation without actually changing the stream. Signature

Stream<T> peek(Consumer<? super T> action)

. peek() operation takes the same argument as the terminal forEach() operation. Think of peek() as an intermediate version of forEach() that returns the original stream back to you.

Stream<String> stream2 = Stream.of("brown-", "bear-");

stream2.sorted().forEach(System.out::print); *// bear-brown*

Stream<String> stream3 = Stream.of("brown bear-", "grizzly-");

stream3.sorted(Comparator.reverseOrder()).forEach(System.out::print); *// grizzly-brown bear*

## Primitive Stream

There are 3 type of Primitive streams

1 **IntStream**: : Used for the primitive types int, short, byte, and char.

2 **LongStream** : : Used for the primitive type long

3 **DoubleStream** : Used for the primitive types double and float

Common primitive stream methods

* **OptionalDouble average()** : The arithmetic mean of the elements. Supported by all primitive Stream type
* **Stream boxed()** : A Stream where T is the wrapper class associated with the primitive value. Supported by all primitive Stream types.
* **OptionalInt max()** : The maximum element of the stream. Supported by integer primitive Stream type
* **OptionalLong max()** : The maximum element of the stream. Supported by long primitive Stream type
* **OptionalDouble max()** : The maximum element of the stream. Supported by double primitive Stream type
* **OptionalInt min()** : The minimum element of the stream. Supported by integer primitive Stream type
* **OptionalLong min()** : The minimum element of the stream. Supported by long primitive Stream type
* **OptionalDouble min()** : The minimum element of the stream. Supported by double primitive Stream type
* **IntStream range(int a, int b)** : Returns a primitive stream from a(inclusive) to b (exclusive)
* **LongStream range(long a, long b)** : Returns a primitive stream from a(inclusive) to b (exclusive)
* **IntStream rangeClosed(int a, int b)** : Returns a primitive stream from a(inclusive) to b (inclusive)
* **LongStream rangeClosed(long a, long b)**: Returns a primitive stream from a(inclusive) to b (inclusive)
* **int sum()** : Returns the sum of the elements in the stream
* **long sum()** : Returns the sum of the elements in the stream
* **double sum()** : Returns the sum of the elements in the stream
* **IntSummaryStatistics summaryStatistics() :** Returns an object containing numerous stream statistics such as the average, min, max, etc.
* **LongSummaryStatistics summaryStatistics()** : Returns an object containing numerous stream statistics such as the average, min, max, etc.
* **DoubleSummaryStatistics summaryStatistics() :** Returns an object containing numerous stream statistics such as the average, min, max, etc.

Create Primitive stream

var random = DoubleStream.generate(Math::random);

var fractions = DoubleStream.iterate(.5, d -> d / 2);

random.limit(3).forEach(System.out::println);

fractions.limit(3).forEach(System.out::println);

FYI Random class provides methods which generate Primitive streams.

Random random = new Random();

random.ints().limit(6).forEach(System.out::println);

Primitive Stream can help in reducing the line of code and complexity drastically.

IntStream count = IntStream.iterate(1, n -> n + 1).limit(5);

count.forEach(System.out::println); *// 12345*

We have to pass 6 as 2nd param is exclusive of last item

IntStream rangeClosed = IntStream.rangeClosed(1, 5);

rangeClosed.forEach(System.out::println); *// 12345*

Since we are using rangeClosed() we can pass 5

## Mapping Stream

We can map streams from one type to another. They have to be compatible types for this to work. Java requires a mapping function to be provided as a parameter



This function takes an Object, which is a String in this case. The function returns an int. The function mappings are intuitive here. They take the source type and return the target type. In this example, the actual function type is ToIntFunction

Stream<String> objStream = Stream.of("penguin", "fish");

IntStream intStream = objStream.mapToInt(s -> s.length());



Primitive Stream can help in reducing the line of code and complexity drastically.

IntStream count = IntStream.iterate(1, n -> n+1).limit(5);

count.forEach(System.out::println); *//12345*

We have to pass 6 as 2nd param is exclusive of last item

IntStream rangeClosed = IntStream.rangeClosed(1, 5);

rangeClosed.forEach(System.out::println); *//12345*

Since we are using rangeClosed() we can pass 5

## Optional Primitives

We have seen Optional can be used for methods which may or may not return value.

var stream = IntStream.rangeClosed(1, 10);

OptionalDouble optional = stream.average();

The OptionalDouble is different from Optional.It is used for Primitive streams. For Wrapper calls we use Optional<Double> and for Primitive class we user OptionalDouble. Working with the primitive optional class looks similar to working with the Optional class itself.The only noticeable difference is that we called getAsDouble() rather than get()

var stream = IntStream.rangeClosed(1, 10);

OptionalDouble optional = stream.average();

optional.ifPresent(System.out::println); *// 5.5*

System.out.println(optional.getAsDouble()); *// 5.5*

var stream2 = IntStream.of();

OptionalDouble optional1 = stream2.average();

System.out.println(optional1.orElseGet(() -> Double.NaN));

What will happen below?

var cats = new ArrayList<String>();

cats.add("Annie");

cats.add("Ripley");

var stream = cats.stream();

cats.add("KC");

System.out.println(stream.count());

We can add values to stream after creation unless its terminated

## Parallel Stream

Parallel Stream takes help of concurrency (multiple threads) to do stream operations. Since it is done using multiple threads hence the sequence is not guaranteed.

List<Integer> ls = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9);

ls.parallelStream().forEach(System.out::print); //653829471 or any random order

ls.parallelStream().sequential().forEach(System.out::print); // Always 123456789

Suppose we have a complex terminal operation which takes about a second to complete for each iteration

public static int transform(int i) {

try {

Thread.sleep(1000);

} catch (Exception e) {

}

return i \* 1;

}

With normal stream it will take 1 second for each item of the stream and it will print sequentially

List<Integer> list = List.of(0,1, 2, 3, 4, 5, 6, 7, 8, 9);

list.stream().map(s -> transform(s)).forEach(System.out::print); //0123456789

But with parallel stream it will complete in 2 seconds with out of order printing

List<Integer> list = List.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

list.parallelStream().map(s -> transform(s)).forEach(System.out::print); //7863914250

If we don’t have control on the source of the stream we can use a parallel method. For example if we are getting our stream from a method like below. We don’t have control if it's parallel or normal.

public static Stream<Integer> getStream() {

List<Integer> list = List.of(0, 1, 2, 3, 4, 5, 6, 7, 8, 9);

return list.stream();

}

We can use the parallel method to make it parallel.

getStream().parallel()

.map(s -> transform(s))

.forEach(x -> System.out.println(x.toString()));

If we add a sequential method at the end it will become sequential again. It's not like a map will run parallel and for each will run sequential. The entire operation will become sequential. The last method before terminal operation will decide how it will run. In this case it is sequential so it will be sequential.

getStream().parallel()

.map(s -> transform(s))

.sequential()

.forEach(x -> System.out.println(x.toString()));

If we put a print statement in worker and then run it parallel we can see that multiple threads from ForkJoinPool is used by parallel method

Thread[main,5,main]

Thread[ForkJoinPool.commonPool-worker-9,5,main]

Thread[ForkJoinPool.commonPool-worker-3,5,main]

Thread[ForkJoinPool.commonPool-worker-13,5,main]

Thread[ForkJoinPool.commonPool-worker-11,5,main]

Thread[ForkJoinPool.commonPool-worker-7,5,main]

Thread[ForkJoinPool.commonPool-worker-5,5,main]

Thread[ForkJoinPool.commonPool-worker-15,5,main]

Thread[main,5,main]

Thread[ForkJoinPool.commonPool-worker-7,5,main]

# Threads

**Multithreading** 🡪 One Java program executed by multiple threads

**Process** 🡪 Any program running on the OS is called process like browser, zoom, word etc. Each process running It is an independent unit that runs in its own memory space. When you open the Chrome browser behind the scenes operating system is going to create a process for the Chrome browser alone. For this process of Chrome browser, the operating system, it is going to allocate some CPU resources, some Ram memory and similarly some disk memory. So whatever hardware resources needed by my process of Chrome, all those will be assigned by the operating system to the process of Chrome. Each process has its own address space and system resources. So whatever address space or resources assigned to Chrome, it cannot be shared or used by other processes like Edge, Safari etc..



**Thread** 🡪 Usually within software or within a program or within a process, you're trying to perform multiple tasks. To handle all these tasks efficiently or parallelly or simultaneously, threads are going to be created within process. A thread is the smallest unit of execution within a process. Multiple threads can exist within a single process, handling the various activities of the software. For example, inside the business logic of Chrome software, someone might have written the code to create a new thread whenever a new tab is being opened by the end user. These threads are going to request the operating system to provide some resources like CPU time or Ram memory. The threads created within a process are going to share the same memory space allowing for efficient communication. The main purpose of creating threads is they are very lightweight compared to the process. And they can run concurrently behind the scenes



Say a CPU has 4 cores so it can process 4 threads at a time. But at any time, we will see many processes running and about 1000s of threads running. This is achieved by allocating a very short duration to each thread in milliseconds, within which a thread may or may not complete its task. And this way threads are rotated based on an algorithm like round robin. This gives an illusion that hundreds of processes and threads are running at once.



Threads approach the operating system to get a CPU time, or to get any other hardware resource that is required for their execution.

the switching from one thread to other thread by the operating system we call that as context switch.

So whenever a thread want to execute a task, the task can be categorized into two categories.

**CPU-bound** All the tasks that require computational power from the CPU, such as Arithmetic operations, logical processing operations, and relational calculations. All these kinds of calculations and operations, since they need CPU power, we can call these kinds of tasks as CPU-bound task.

**IO-bound tasks** are highly dependent on input output operations. For example, if you're trying to play a YouTube video inside your Chrome browser, it is not a CPU-bound task, it is a IO-bound task because the browser in this scenario it does not require the CPU time instead, it is going to rely on the network of your system to load the video from a remote server. Any task which involves communication over a network or reading or writing operations onto a file or onto a remote server. All these kinds of tasks we can consider them as IO-bound task. And while IO task is being executed, the thread is not going to do any job. It is going to sit ideally behind the scenes because IO-bound tasks are going to be executed based upon the network of your system.



A group of rectangular signs with text

Description automatically generated

Create thread – Threads can be created using 2 ways either by extending Thread class or by implementing Runnable interface

Thread class has run method we can extend the thread class and override the run method and then with start method from Thread class we can create a new thread.

public class Test extends Thread  {

    public static void main(String[] args) {

        System.out.println("Main" +"--"+Thread.currentThread().getName());      *// Will be done by main thread*

        Test test = new Test();

        test.start();

    }

    @Override

    public void run(){

        for (int i = 0; i < 4; i++) {

            //Will be done by new thread

            System.out.println("Task from thread "+ i +"--"+Thread.currentThread().getName());

        }

    }

}

Output

Main--main

Task from thread 0--Thread-0

Task from thread 1--Thread-0

Task from thread 2--Thread-0

Task from thread 3--Thread-0

Sometimes our base class already extends some other class in that case we can’t extend another class. In that case we can implement Runnable interface override the run method and pass the instance to thread class to create new thread.

public class Test implements Runnable  {

    public static void main(String[] args) {

        System.out.println("Main" +"--"+Thread.currentThread().getName());      *// Will be done by main thread*

        Test test = new Test();

        Thread thread = new Thread(test);

        thread.start();

    }

    @Override

    public void run(){

        for (int i = 0; i < 4; i++) {

            //Will be done by new thread

            System.out.println("Task from thread "+ i +"--"+Thread.currentThread().getName());

        }

    }

}

Main--main

Task from thread 0--Thread-0

Task from thread 1--Thread-0

Task from thread 2--Thread-0

Task from thread 3--Thread-0

If we see the thread class it has a constructor which accepts runnable.

public Thread(Runnable target) {  
 this(null, target, "Thread-" + nextThreadNum(), 0);  
}

Runnable is a functional interface with single abatract method run with no return type and no parameter

@FunctionalInterface  
public interface Runnable {  
  
 public abstract void run();  
}

Which means we can pass a lambda expression to thread constructor

public class Test {

    public static void main(String[] args) {

        Thread thread = new Thread(

                () -> System.out.println("Thread from Lambda" + "--" + Thread.currentThread().getName()));

        thread.start();

        System.out.println("Main" + "--" + Thread.currentThread().getName()); *// Will be done by main thread*

    }

}

Main--main

Thread from Lambda--Thread-0

getId getName method

Thread t1 = new Thread();  
Thread t2 = new Thread();  
Thread t3 = new Thread();  
t3.setName("MyThread");  
  
System.out.println("Thread 1 ID is : " + t1.getId() + ", name : "+ t1.getName());  
System.out.println("Thread 2 ID is : " + t2.getId() + ", name : "+ t2.getName());  
System.out.println("Thread 3 ID is : " + t3.getId() + ", name : "+ t3.getName());

curerntThread method

Thread mainThread = Thread.currentThread();  
System.out.println(mainThread.getId());  
System.out.println(mainThread.getName());

Thread sleep method

long startTime = System.currentTimeMillis();  
for(int i=0;i<5;i++){  
 System.out.println("Hello from Main method : " + i);  
 Thread.sleep(1000);  
}  
long endTime = System.currentTimeMillis();  
System.out.println("Total time taken : " + (endTime-startTime));

**Thread.yield** – It’s a hint to CPU that other threads can be prioritized and this can wait, however CPU can ignore the request

class Task2 implements Runnable {

    public void run() {

        Thread.yield();

        for (int i = 51; i < 100; i++) {

            System.out.println("Task 2 -> "+ i);

        }

    }

}

**Thread join** - The join() method in Java is provided by the java.lang.Thread class and allows one thread to wait until another thread completes its execution. This method is particularly useful in multithreading scenarios where you need to ensure that a particular thread has finished its task before proceeding with the next one.

E.g. below until custom thread completes the main thread will not execute

    public static void main(String[] args) {

        Thread threadA = new Thread(

                () -> {

                    for(int i=1;i<=5;i++) {

                        System.out.println("Thread A - count : "+i);

                        try {

                            Thread.sleep(500);

                        } catch (InterruptedException e) {

                            throw new RuntimeException(e);

                        }

                    }

                }

        );

        threadA.start();

        try {

            threadA.join();

        } catch (InterruptedException e) {

            throw new RuntimeException(e);

        }

        System.out.println("Main thread ended");

    }

**setPriority()** Thread priority  you can set the priority of a thread using the setPriority method of the Thread class. Thread priorities are integers ranging from Thread.MIN\_PRIORITY (which is 1) to Thread.MAX\_PRIORITY (which is 10). The default priority is Thread.NORM\_PRIORITY (which is 5). Thread priority is just a hint to the thread scheduler. It doesn’t guarantee the order of execution based on priority. The actual behavior can vary depending on the JVM implementation and the underlying operating system. If the threads have same priority it will be executed in FIFO order

    public static void main(String[] args) {

        Thread thread1 = new Thread(() -> System.out.println("Thread1"));

        Thread thread2 = new Thread(() -> System.out.println("Thread2"));

        Thread thread3 = new Thread(() -> System.out.println("Thread3"));

        thread1.setPriority(Thread.MIN\_PRIORITY);

        thread2.setPriority(Thread.NORM\_PRIORITY);

        thread3.setPriority(Thread.MAX\_PRIORITY);

        thread1.start();

        thread2.start();

        thread3.start();

    }

**wait, notify notifyAll** The wait() and notify() methods are used for inter-thread communication. These methods are part of the Object class and are used to manage the synchronization of threads.

* **wait()**: This method causes the current thread to wait until another thread invokes the notify() or notifyAll() method for the same object. The thread releases the lock on the object and enters the waiting state.
* **notify()**: This method wakes up a single thread that is waiting on the object's monitor. If multiple threads are waiting, one of them is chosen to be awakened.
* **notifyAll()**: This method wakes up all the threads that are waiting on the object's monitor.

Say we have a class SharedResource with 2 methods producer and consumer and we want consumer to fire only once after producer has produced something. We do this with Boolean available,

when true 🡪 producer won’t emit and will wait

consumer will consume and notify

when false 🡪 producer will produce

consumer won’t consume and will wait

class SharedResource {

    private boolean available = false;

    public synchronized void produce() throws InterruptedException {

        while (available) {

            wait();

        }

        System.out.println("Producing resource...");

        available = true;

        notify();

    }

    public synchronized void consume() throws InterruptedException {

        while (!available) {

            wait();

        }

        System.out.println("Consuming resource...");

        available = false;

        notify();

    }

}

Now when main method is triggered it will start produce in one thread, and then consumer in another. When first started available is false so it will not wait and produce change available to false and trigger notify which will start consumer and the process will repeat ensuring after one produce one consume is executed.

public class Test {

    public static void main(String[] args) {

        SharedResource resource = new SharedResource();

        Thread producer = new Thread(() -> {

            try {

                for (int i = 0; i < 5; i++) {

                    resource.produce();

                    Thread.sleep(1000);

                }

            } catch (InterruptedException e) {

                Thread.currentThread().interrupt();

            }

        });

        Thread consumer = new Thread(() -> {

            try {

                for (int i = 0; i < 5; i++) {

                    resource.consume();

                    Thread.sleep(1000);

                }

            } catch (InterruptedException e) {

                Thread.currentThread().interrupt();

            }

        });

        producer.start();

        consumer.start();

    }

}

**volatile** In a multi-threaded application, each thread may have its own local copy of a variable. When a thread updates a variable, the change might not be immediately visible to other threads. This can lead to inconsistent data and unpredictable behaviour. The volatile keyword ensures that any write to a volatile variable is immediately visible to all threads The volatile keyword in Java is a modifier used to ensure that the value of a variable is always read from and written to the main memory, rather than being cached locally by individual threads.

In below example we are tracking the status of stopFlag Boolean in one thread and changing the status in a different thread. With volatile updated value will reflect instantly as seen from current milliseconds. In local we may not see the difference but a system with complicated logic and more CPU load without volatile will see the difference.

class SharedBooleanResource {  
  
 private volatile boolean stopFlag = false;  
  
 public void doWork() {  
 while (!stopFlag) {  
 System.out.println("Working...--> " +System.currentTimeMillis());  
 }  
 System.out.println("Work stopped--> " +System.currentTimeMillis());  
 }  
  
 public void setStopFlag() {  
 System.out.println("Changing flag to false --> " +System.currentTimeMillis());  
 stopFlag = true;  
 }  
}  
  
public class Test {  
 public static void main(String[] args) throws InterruptedException {  
 SharedBooleanResource sharedBooleanResource = new SharedBooleanResource();  
 Thread workerThread = new Thread(sharedBooleanResource::doWork);  
  
 Thread stopperThread = new Thread(() -> {  
 try {  
 Thread.sleep(2000);  
 sharedBooleanResource.setStopFlag();  
 } catch (InterruptedException e) {  
 throw new RuntimeException(e);  
 }  
 });  
 workerThread.start();  
 stopperThread.start();  
 workerThread.join();  
 stopperThread.join();  
 }  
}

**Deadlock** Deadlock is a scenario where one of the threads is waiting to acquire the lock of an object currently held by another thread. For e.g. thread, t1, it is waiting for a object which is already locked by another thread, t2, and the second thread is waiting to obtain the lock of a different object held by the first thread. So consequently, both these threads find themselves in a state of mutual waiting. Expecting each other will release the lock and this will result into a deadlock situation.



Different states of Thread

1. New 🡪 created , start method not called yet not running
2. Runnable 🡪 Ready to run but not running now, other thread running
3. Running 🡪 Running
4. Blocked/waiting 🡪Blocked as called some external database or interface and waiting for it to complete or waiting for data from some other thread
5. Terminated 🡪 Completed execution

## Executer Service

Normal way of starting thread had a few disadvantages like

1. Code for starting new thread is cumbersome and has a lot of boilerplate code
2. No control on how many threads can be started, can’t place any restriction.
3. Can’t put any condition like start new thread when either of the existing thread is completed. For e.g. if we have 2 thread- Therad1 and Thread2, we can put condition to start new thread only once Thread1 and Thread2 is completed individually, but can’t put condition that when either of Thread1 or Thread2 is completed start new thread
4. Can’t return value from thread.

These can be overcome using Executer service implementation below. We create an instance of ExecutorService using Executors.newFixedThreadPool(2) method and pass the Task in in the execute method of ExecutorService

Since we have mentioned newFixedThreadPool(2) so only 2 threads will run at a time once they complete then new Threads will start

public class Test {

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

        ExecutorService exe = Executors.newFixedThreadPool(2);

        // Once 2 threads are completed then only new Threads are started

        exe.execute(new Task(1));

        exe.execute(new Task(2));

        exe.execute(new Task(3));

        exe.execute(new Task(4));

        exe.shutdown();

    }

}

class Task extends Thread {

    private int count;

    Task(int count) {

        this.count = count;

    }

    public void run() {

        System.out.println("Task " + count + " started");

        for (int i = 0; i < 5; i++) {

            System.out.println("Task-" + count + " " + i);

        }

        System.out.println("Task " + count + " ended");

    }

}

Return from Thread using executor service. Instead of execute method we will use submit method , which will return a Future from where we can get the result using get() method.

Main will run only when ExecutorService thread has completed.

public class Test {

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

        ExecutorService executorService = Executors.newFixedThreadPool(1);

        Future<String> future = executorService.submit(new CallableTask("Amit"));

        //Future holds a promise

        System.out.println(future.get());

        System.out.println("Main completed");

        executorService.shutdown();

    }

}

class CallableTask implements Callable<String> {

    private String greeting;

    CallableTask(String st) {

        this.greeting =st;

    }

    @Override

    public String call() throws Exception {

        Thread.sleep(1000);

        return "Hello" +greeting;

    }

}

Multiple tasks 🡪 Here we can initiate multiple tasks using ExecutorService. We create a list of tasks and pass to ExecutorService and invoke all using invokeAll() method and retrieve the results

public class Test {

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

        ExecutorService executorService = Executors.newFixedThreadPool(3);

        List<CallableTask> callableTasks = List.of(new CallableTask("Amit",1000),

                new CallableTask("Shamit",500),

                new CallableTask("Bhamit",2000));

        List<Future<String>> future = executorService.invokeAll(callableTasks);

        for (Future<String> result : future) {

            System.out.println(result.get());

        }

        System.out.println("Main completed");

        executorService.shutdown();

    }

}

class CallableTask implements Callable<String> {

    private String greeting;

    private int wait;

    CallableTask(String st, int wait) {

        this.greeting = st;

        this.wait = wait;

    }

    @Override

    public String call() throws Exception {

        Thread.sleep(this.wait);

        return "Hello" + greeting;

    }

}

In above example even tough different tasks are having different wait time, Still we get the results when all the ExecutorService has completed. We can overcome this by calling invokeAny method of ExecutorService. This will return as soon as the results are available but it will return the first result and discard others. It doesn’t wrap the result in Future but just returns the result.

public class Test {

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

        ExecutorService executorService = Executors.newFixedThreadPool(3);

        List<CallableTask> callableTasks = List.of(new CallableTask("Amit", 1000),

                new CallableTask("Shamit", 500),

                new CallableTask("Bhamit", 2000));

        String future = executorService.invokeAny(callableTasks);

        System.out.println(future);

        System.out.println("Main completed");

        executorService.shutdown();

    }

}

class CallableTask implements Callable<String> {

    private String greeting;

    private int wait;

    CallableTask(String st, int wait) {

        this.greeting = st;

        this.wait = wait;

    }

    @Override

    public String call() throws Exception {

        Thread.sleep(this.wait);

        return "Hello" + greeting;

    }

}

## Virtual Threads

**Introduction** The latest buzzword in Java multithreading is virtual threads. Introduced in Java 21, virtual threads are poised to revolutionize applications that utilize a large number of threads as part of their business logic.

**What Are Virtual Threads?** Virtual threads are a new feature in Java 21. They allow for the creation of millions of threads within a Java program without the resource constraints typical of traditional threads. Unlike traditional threads, which are limited by RAM, memory, and CPU cores, virtual threads are lightweight and can scale significantly better.

Key Differences

* **Traditional Threads:** Heavyweight, limited by system resources.
* **Virtual Threads:** Lightweight, allowing the creation of millions of threads without the same constraints.

**Background and Development** The Oracle team has been working on virtual threads for many years under Project Loom. Project Loom aims to simplify the development, maintenance, and monitoring of high-throughput concurrent applications. After extensive research and testing, virtual threads were introduced in Java 21.

**Understanding Threads in Operating Systems** To fully grasp virtual threads, we need to understand the basics of operating system threads. There are two types of threads in operating systems:

1. **Kernel Threads**: Also known as operating system threads, kernel threads are managed and scheduled by the OS kernel. They are resource-intensive, requiring system calls for creation, scheduling, and synchronization.
2. **User Threads**: Created by applications and managed at the user level. They are lightweight compared to kernel threads and are created and destroyed quickly.

**Thread Models** The mapping between user threads and kernel threads can be done using three models:

1. **M:1 Model:** Many user threads are mapped to a single kernel thread.
2. **1:1 Model:** Each user thread is mapped to a single kernel thread.
3. **M:N Model:** Many user threads are mapped to a pool of kernel threads.

**Implications for Java** In Java, the application starts with a default thread to run the main() method. Additional user threads can be created as needed. These user threads are executed by kernel threads, meaning each user thread must be mapped to a kernel thread to get CPU time.

**Evolution of Threading** in Java Java has evolved to support different threading models, and virtual threads represent the latest advancement. By using lightweight virtual threads, Java programs can achieve higher concurrency without the performance bottlenecks associated with traditional threads.



Early Days of Java: Green Threads

When Java first appeared in the 1990s, it used a thread model known as green threads. These green threads were implemented entirely in the Java Virtual Machine (JVM) and mapped multiple user-level threads to a single kernel thread, following a many-to-one model.

Key Characteristics of Green Threads:

* **Execution and Scheduling**: Green threads were managed by the JVM, which was responsible for scheduling and overseeing these threads.
* **Performance**: While green threads were fast in terms of creation and execution, they had significant drawbacks:
  + **Scalability**: Java could not effectively leverage multiple CPU cores. Regardless of the number of cores, all user threads were mapped to a single kernel thread.
  + **Complexity**: Libraries or programs needed to know low-level JVM details, increasing the learning curve and complexity.

Limitations of Green Threads:

* **Single-Core Limitation**: During the 1990s, CPUs typically had a single core. As hardware advanced and multi-core processors became standard, the green thread model could not take advantage of multiple cores.
* **JVM Overhead**: The JVM's responsibility for mapping and scheduling threads added overhead and limited scalability.

Due to these limitations, green threads were phased out after Java 1.1. From Java 1.2 onwards, Java introduced native threads.

**Native Threads (Platform Threads)** Starting from Java 1.2, Java adopted a one-to-one thread model, where each user thread maps directly to a kernel thread. These threads are known as native threads or platform threads.

Key Characteristics of Native Threads:

* **Mapping**: Each user thread corresponds to a single kernel thread.
* **Management**: Managed by the JVM with the operating system's assistance.
* **Performance**: High efficiency during execution, but with notable costs for starting and stopping threads.

Limitations of Native Threads:

* **Resource Intensive**: Creating and maintaining native threads is resource-intensive. Each thread requires memory for its call stack and local memory, ranging from 1MB to 20MB.
* **Limited Scalability**: The number of threads that can be created is limited by the available RAM. For example, on a system with 8GB of RAM, creating 1 million native threads is impractical.
* **Context Switching**: Context switching between threads involves system calls to the kernel, which is an expensive operation.

Despite these limitations, native threads have been the standard from Java 1.2 to Java 20.

**Virtual Threads: The Future of Java Concurrency**With Java 21, a new type of thread model, known as virtual threads, was introduced. Virtual threads aim to address the limitations of native threads and provide a more scalable and efficient threading model.

Advantages of Virtual Threads:

* **Lightweight**: Virtual threads are lightweight and can be created in large numbers without significant memory overhead.
* **Efficient Resource Utilization**: Virtual threads utilize CPU resources more efficiently, minimizing idle time during I/O operations.
* **Scalability**: Virtual threads allow for the creation of millions of threads, making it easier to scale applications and handle large numbers of concurrent tasks.

Example Scenario: Web Server with Native Threads vs. Virtual Threads

Consider a web server with 16GB of RAM running a web application. Each incoming request is handled by a separate thread. With native threads, creating 1600 threads (assuming each thread requires 10MB of RAM) utilizes all available memory, limiting the server's capacity to handle concurrent requests.

However, with virtual threads, the server can handle significantly more requests. Since virtual threads are lightweight, they do not consume as much memory, allowing for the creation of millions of threads. This scalability improves the server's ability to handle high loads and increases overall performance.

**Conclusion** Java's thread model has evolved from green threads in the early versions to native threads and now virtual threads. Each transition has addressed the limitations of the previous model, resulting in more efficient and scalable concurrency mechanisms. Virtual threads, introduced in Java 21, promise to revolutionize how Java applications handle concurrency, making it easier to build scalable and high-performance applications.

Java 21 introduces virtual threads, a significant enhancement over traditional platform or native threads. Let's explore the advantages of virtual threads and how they work.

Advantages

1. Lightweight:
   * Virtual threads are extremely lightweight, consuming only a few bytes of memory.
   * In contrast, platform threads require between 1MB to 20MB of memory, sometimes even more.
   * This drastic reduction in memory footprint allows for the creation of a vast number of virtual threads without overwhelming the system's RAM.
2. Scalability:
   * Due to their minimal memory requirements, you can create millions of virtual threads.
   * This scalability is a game-changer, enabling applications to handle massive concurrency without significant overhead.
3. No System Calls for Creation:
   * Creating a virtual thread does not require a system call to the kernel.
   * The mapping between a virtual thread and the kernel thread is managed indirectly, reducing the overhead associated with thread creation.
4. Efficient Creation and Destruction:
   * Virtual threads can be created and destroyed extremely quickly, often in nanoseconds.
   * This efficiency further enhances the system's ability to handle high concurrency.
5. Enhanced CPU Utilization:
   * Virtual threads utilize the M:N threading model, where many virtual threads are mapped to a smaller number of OS threads.
   * JVM creates a pool of platform threads at startup, which are then used to execute virtual threads.
   * When a virtual thread performs an IO operation, it detaches from its platform thread, allowing the platform thread to execute another virtual thread. This reduces idle time and improves CPU utilization.
6. Reduced Context Switching Overhead:
   * Context switching between virtual threads is managed by the JVM and does not involve system calls.
   * This reduction in overhead improves the performance and responsiveness of applications.

How Virtual Threads Work

1. JVM Initialization:
   * Upon startup, the JVM creates a set of platform threads.
   * These platform threads are mapped to OS threads and are ready to execute virtual threads.
2. Creation of Virtual Threads:
   * Developers can create numerous virtual threads within the Java heap memory, thanks to their lightweight nature.
3. Execution and Management:
   * The JVM manages the mapping of virtual threads to platform threads.
   * When a virtual thread is assigned to a platform thread, its instructions are executed promptly.
   * After execution, the virtual thread can be destroyed or kept in a waiting state if it is waiting for an IO operation.
4. Handling IO Operations:
   * During IO operations, virtual threads detach from their platform threads.
   * This allows the platform threads to be reassigned to other virtual threads that need CPU time, maximizing CPU utilization.

Conclusion

Virtual threads in Java 21 offer a robust solution for handling high concurrency with minimal overhead. Their lightweight nature, scalability, efficient creation and destruction, enhanced CPU utilization, and reduced context switching overhead make them a powerful tool for modern Java applications.

## Virtual Threads in Java 21: A Demonstration

### Inside This Lecture

In this lecture, we'll demonstrate the creation and performance of virtual threads in Java 21. We'll compare the performance of virtual threads with traditional platform threads.

### Creating the VirtualThreadDemo Class

1. Main Method:
   * Start by creating a Java class named VirtualThreadDemo.
   * Define the main() method.
2. Runnable Object:
   * Create a Runnable object using a lambda expression.
   * This Runnable will generate two random double values, multiply them, and store the result in a variable.

java

Copy code

import java.util.Random;

public class VirtualThreadDemo {

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

Random random = new Random();

Runnable runnable = () -> {

double result = random.nextDouble() \* 1000 \* random.nextDouble() \* 1000;

System.out.println(result);

};

// Measure time for normal threads

long startTime = System.currentTimeMillis();

for (int i = 0; i < 500000; i++) {

Thread thread = new Thread(runnable);

thread.start();

thread.join();

}

long endTime = System.currentTimeMillis();

System.out.println("Total time for normal threads: " + (endTime - startTime) + " ms");

// Measure time for virtual threads

startTime = System.currentTimeMillis();

for (int i = 0; i < 500000; i++) {

Thread vThread = Thread.startVirtualThread(runnable);

vThread.join();

}

endTime = System.currentTimeMillis();

System.out.println("Total time for virtual threads: " + (endTime - startTime) + " ms");

}

}

### Explanation

1. Runnable Object:
   * A Runnable object is created to generate two random double values, multiply them, and print the result.
2. Normal Threads:
   * A loop creates and starts 500,000 normal threads.
   * Each thread executes the Runnable task.
   * The join() method ensures the main thread waits for each thread to complete before starting the next one.
   * The time taken to execute all threads is measured and printed.
3. Virtual Threads:
   * The same loop creates and starts 500,000 virtual threads using Thread.startVirtualThread(runnable).
   * The time taken to execute all virtual threads is measured and printed.

### Output

* **Normal Threads**: The program typically takes around 30 seconds to complete.
* **Virtual Threads**: The program completes in approximately 9 seconds, demonstrating the efficiency of virtual threads.

### Key Points

* **Memory Efficiency**: Virtual threads consume significantly less memory compared to platform threads.
* **Performance**: Virtual threads offer improved performance due to lower overhead in creation and management.
* **Static Methods**: Java 21 provides static methods like Thread.startVirtualThread() and Thread.ofVirtual() to create virtual threads easily.

### Additional Methods

* **Thread.isVirtual()**: This static method checks if a given thread is a virtual thread.

### Conclusion

Virtual threads in Java 21 provide a powerful tool for building highly concurrent applications with minimal overhead. They offer significant performance improvements and are easier to use with familiar threading constructs.

# Questions

1. Is Java Software platform dependent?

Yes

1. Is both i and J are initialized int i,j = 8;

No only J

1. Why Java gives 2 bytes 2 ^16 size to char

To accommodate all special characters of various languages

1. How can we use characters not in key board?

Char uses Unicode so we can use Unicode

char i = '\u5632';

1. Can we add char

Yes

1. What is constructor overloading ?
2. Difference between methods and constructors?

Methods can be called only once constructors are called only when class is initialized

Methods can have any name Constructors only have class name

Methods can have return type Constructors don’t have

1. Is the constructor called when we declare a Class variable?

No Only when initialized

Alpha alpha; or Alpha alpha = null;

Only with new key word constructor is called

alpha = new Alpha();

1. What is the use of Instance Initializers or code blocks.

Needed for Anonymous class which don’t have constructors to handle initialiozation logic.

Also if you have any common logic for multiple constructor you can keep in code blocks

1. Instance Initializers or code blocks overwrite Constructors?

No they are implemented before Constructors

1. Can a class implement 2 interface with same methods?

Yes it has to implement the methods

1. What is native method?

It marks a method, that it will be implemented in other languages, not in Java. It works together with JNI (Java Native Interface).

[**https://stackoverflow.com/questions/6101311/what-is-the-native-keyword-in-java-for**](https://stackoverflow.com/questions/6101311/what-is-the-native-keyword-in-java-for)

1. What is clone method in Java?

It is a method in Object class to create shallow clone i.e. same values but different address

It is protected method

The call implementing but implement cloneable interface

class Alpha implements Cloneable{  
  
 public Object clone() throws CloneNotSupportedException {  
 return super.clone();  
 }  
 int beta;  
  
 public Alpha(int beta) {  
 this.beta = beta;  
 }  
  
}

Alpha a = new Alpha(3);  
Alpha ab = (Alpha) a.clone();  
System.*out*.println(a == ab); //false  
System.*out*.println(a.equals(ab)); //false  
System.*out*.println(a.beta +" "+ab.beta); //3 3  
a.beta = 7;  
System.*out*.println(a.beta +" "+ab.beta); //7 3

1. What are different classes and interfaces in collections?

List

Set

Queue

Blocking Queue

Map

1. Difference between ArrayList and Linked List

ArrayList implements array all items are indexed, List uses internal implementation where each item points to previous and next items

ArrayList is good for sequential addition, random access due to indexing . Linked list is good when we add items in between as we just have to change the pointers of previous and next item for Array list we have to shift all elements.

When we create once and do frequent reads Array list is best.

1. Difference between Arrray list/ Vectors and Hashtable VS Hashmaps

Vectors and Hashtables are synchronized and hence slow

1. Difference between Hashmap and Linked HashMap

LinkedHashmap maintains order in which they were added, Hashmap doesn’t do it

1. Default capacity of ArrayList

10

1. Default capacity of HashMap

16 , Loadfactor 75%

1. Comparable vs comparator

Comparable ¦ Comparator ¦

¦-----------------------------------------+------------------------------------------¦

¦ java.lang.Comparable ¦ java.util.Comparator ¦

¦-----------------------------------------+------------------------------------------¦

¦ int objOne.compareTo(objTwo) ¦ int compare(objOne, objTwo) ¦

¦-----------------------------------------+------------------------------------------¦

¦ Negative, if objOne < objTwo ¦ Same as Comparable ¦

¦ Zero, if objOne == objTwo ¦ ¦

¦ Positive, if objOne > objTwo ¦ ¦

¦-----------------------------------------+------------------------------------------¦

¦ You must modify the class whose ¦ You build a class separate from to sort. ¦

¦ instances you want to sort. ¦ the class whose instances you want ¦

¦-----------------------------------------+------------------------------------------¦

¦ Only one sort sequence can be created ¦ Many sort sequences can be created ¦

¦-----------------------------------------+------------------------------------------¦

¦ Implemented frequently in the API by: ¦ Meant to be implemented to sort ¦

¦ String, Wrapper classes, Date, Calendar ¦ instances of third-party classes.