## JAVA

Q). Why java is platform independent ?

The key factor that makes Java platform-independent is the Java Virtual Machine (JVM). Here's how it works:

1. Compilation to Bytecode: When you write Java code, it is compiled by the Java compiler into bytecode, not machine code. This bytecode is stored in .class files. Unlike native code (such as C or C++), which is compiled for a specific operating system and hardware architecture, Java bytecode is universal.
2. Execution by the JVM: The bytecode is executed by the Java Virtual Machine (JVM), which is a software-based interpreter. The JVM is available for different platforms (Windows, Linux, macOS, etc.), and each platform has its own specific JVM implementation that translates bytecode into machine code that the platform can understand and execute.
3. Write Once, Run Anywhere: Since the bytecode is platform-agnostic, you can run the same Java application on any system that has a compatible JVM. As long as the JVM for that platform exists, the bytecode can be interpreted and executed, regardless of the underlying hardware or operating system. This is the core of Java's platform independence.

--------------------------------------------------------------------------------------------------------------

Q). Which class is always present in every class ?

In every Java class, the Object class is always present, either directly or indirectly. This is because all classes in Java implicitly inherit from the Object class, which is the root class of the entire Java class hierarchy.

Key points about the Object class:

* Super class of every class: Every class in Java, except Object itself, directly or indirectly extends the Object class. Even if you don't explicitly declare a parent class, your class inherits from Object by default.
* Methods in Object: The Object class defines several important methods, which all classes inherit. These include:
  + toString(): Returns a string representation of the object.
  + equals(): Compares two objects for equality.
  + hashCode(): Returns a hash code for the object.
  + clone(): Creates and returns a copy of the object (though it requires the class to implement Cloneable).
  + getClass(): Returns the Class object that represents the class of the object.
  + notify(), notifyAll(), and wait(): These are used for thread synchronization.

Thus, the Object class is the superclass of every other class in Java, ensuring that every class inherits these fundamental methods.

--------------------------------------------------------------------------------------------------------------

Q). What are the four pillars of OOPS ?

The four pillars are: APIE

Abstraction

* Definition: Abstraction involves hiding the complex implementation details and showing only the essential features of an object. It allows a programmer to focus on what an object does rather than how it achieves it.
* Benefit: Abstraction simplifies the interaction with complex systems, improving readability, usability, and maintainability of the code.

Polymorphism

* Definition: Polymorphism allows one interface to be used for a general class of actions. The specific action is determined by the type of object it is acting upon. In OOP, polymorphism allows methods to do different things based on the object it is acting upon.
* Types of Polymorphism:
  + Compile-time polymorphism (Method Overloading): The same method name is used with different parameters.
  + Run-time polymorphism (Method Overriding): A method in the subclass overrides a method in the superclass to provide specific behavior.
* Benefit: Polymorphism allows you to write flexible and reusable code that can work with objects of different types.

Inheritance

* Definition: Inheritance allows one class (the subclass) to inherit the properties and behaviors (fields and methods) of another class (the superclass). This facilitates code reuse and establishes a natural hierarchy.
* Benefit: Inheritance promotes code reuse, reducing redundancy, and makes it easier to manage and extend functionality by creating specialized versions of a class.

Encapsulation

* Definition: Encapsulation is the concept of bundling data (attributes) and methods (functions) that operate on the data into a single unit or class. It also involves restricting direct access to some of an object's components, typically by making attributes private and providing public getter and setter methods to access them.
* Benefit: Encapsulation helps to protect the internal state of an object by restricting direct access and modifying it only through controlled methods, ensuring better data integrity and security.

----------------------------------------------------------------------------------------------------------------------

Q). What is the Super keyword?

The super keyword in Java is a reference variable used to refer to the parent class (superclass) of the current object. It is often used in inheritance to access members (fields, methods) of the parent class, and it serves a few key purposes.

Key Uses of the super Keyword:

Accessing Parent Class Constructor:

* + The super() keyword is used to call the constructor of the parent class. It must be the first statement in the subclass constructor.
  + If you do not explicitly call a constructor of the parent class, Java automatically calls the default constructor of the parent class (if one exists).

Accessing Parent Class Methods:

You can use super to call a method from the parent class, especially if the method is overridden in the subclass and you want to call the parent class's version.

Accessing Parent Class Fields:

If a field in the parent class is hidden by a field of the same name in the subclass, you can use super to refer to the parent class's field.

Summary:

super() is used to call the parent class's constructor.

super.methodName() is used to invoke a method from the parent class.

super.fieldName is used to access a field from the parent class when it is hidden by a subclass field.

In essence, the super keyword allows the subclass to access and interact with the parent class, enabling more flexible and controlled inheritance.

--------------------------------------------------------------------------------------------------------------

Q). Why Java is Preferring OOPS concept?

1. Modularity and Code Organization

* OOP in Java promotes modularity by organizing the code into small, self-contained units called classes. Each class can have its own data (attributes) and methods (behaviors), making it easier to understand and manage large codebases.
* Classes allow programmers to design complex systems as a collection of simpler, logically structured parts. This makes debugging and maintaining code easier because each class can be developed and tested independently.

2. Reusability

* Java promotes code reuse through inheritance. With inheritance, Java allows one class (a subclass) to inherit fields and methods from another class (a superclass), minimizing code duplication.
* The Object-Oriented approach allows developers to build libraries of classes that can be reused across multiple projects. Once a class is written and tested, it can be reused without modification, which speeds up the development process.

3. Maintainability

* Encapsulation (another core principle of OOP) helps improve the maintainability of Java applications. By hiding the internal workings of an object and providing controlled access through public methods (getters/setters), Java makes it easier to change the internal implementation without affecting other parts of the system.
* If a bug arises or a new feature needs to be added, the impact of changes is limited to specific classes, making maintenance easier.

4. Scalability

* Concepts like polymorphism and inheritance allow developers to extend the existing codebase without disrupting the existing system. This flexibility makes Java applications highly scalable.

5. Abstraction for Simplification

* Abstraction, a key feature of OOP, allows developers to hide complex details and provide only essential features to users. This simplifies the interface of classes and objects and makes systems more user-friendly.
* By focusing on what an object does instead of how it achieves it, developers can design high-level systems and improve system usability and efficiency.

6. Real-World Modeling

* Java’s OOP model is a natural fit for modeling real-world entities, as it uses objects that represent real-world concepts. This makes it easier for developers to design systems that are more intuitive and aligned with how people think about the problem domain.
* For example, objects like "Car," "Employee," or "Account" can be easily modeled using Java classes, making it easier for developers to translate real-world concepts into software solutions.

7. Security

* OOP in Java also enhances security through encapsulation. By restricting access to an object's internal state and exposing only necessary functionality, Java helps ensure that objects cannot be tampered with directly, which reduces the risk of unintended changes or security vulnerabilities.
* Java also allows for controlled access to class members using access modifiers (private, protected, public), which helps implement robust security features in applications.

8. Polymorphism for Flexibility

* Polymorphism, another key OOP concept, allows one interface to be used for different underlying forms (objects). This flexibility allows developers to use the same code to work with different types of objects in a generalized manner.

9. Support for Distributed Computing

* Java’s OOP model works well with distributed computing environments. Features like remote method invocation (RMI) and web services allow Java applications to interact with distributed objects, making it suitable for building enterprise-level applications that run on multiple machines.

10. Cross-Platform Compatibility

* Java's OOP features, combined with its architecture (using the Java Virtual Machine (JVM)), ensure that Java code is platform independent. Java programs are compiled into bytecode, which can run on any platform that supports the JVM, making it a truly cross-platform programming language.
* This is aligned with Java's "Write Once, Run Anywhere" (WORA) philosophy, which is enabled by its object-oriented structure.

Conclusion:

Java’s preference for OOP is driven by the numerous benefits it provides, such as modularity, reusability, maintainability, scalability, and the ability to model real-world problems effectively. By leveraging the core OOP principles of encapsulation, abstraction, inheritance, and polymorphism, Java allows developers to create flexible, modular, secure, and easily maintainable applications. These advantages make Java one of the most popular and powerful programming languages for developing a wide range of applications.

--------------------------------------------------------------------------------------------------------------

Q). What are errors and Exceptions?

In Java, errors and exceptions are both types of Throwable objects that represent problems or unusual conditions that occur during program execution. However, they have different causes and handling mechanisms. Here's a breakdown of errors and exceptions:

* 1. Errors
* Definition: Errors are serious issues that typically occur during the execution of a program and are usually beyond the control of the programmer. They represent problems related to the environment in which the program is running.
* Characteristics:
  + Errors are generally not intended to be caught or handled by the program. They often indicate critical problems that make it impossible for the program to continue.
  + Errors typically occur due to system-level issues like hardware failure, JVM failure, or other unrecoverable conditions.
* Examples of Errors:
  + OutOfMemoryError: This occurs when the JVM runs out of memory to allocate for objects.
  + StackOverflowError: This happens when a program recurses too deeply, causing the stack to overflow.
  + VirtualMachineError: This occurs when there are errors in the Java Virtual Machine itself, such as the JVM crashing.
* Handling Errors: Errors are not usually caught using a try-catch block because they are considered fatal and should not be handled in normal application logic.
  1. Exceptions
* Definition: Exceptions are conditions or events that occur during the execution of a program and disrupt its normal flow. Unlike errors, exceptions are usually caused by programmatic issues or external events like incorrect user input, file I/O issues, or network failures.
* Characteristics:
  + Exceptions can often be handled using try-catch blocks or declared to be thrown with the throws keyword.
  + Exceptions are typically due to programmable conditions (like invalid inputs or faulty logic), and many can be recovered from or prevented through proper handling.
* Types of Exceptions: Java exceptions are divided into two main categories:

Checked Exceptions:

* + - These are exceptions that the compiler requires you to handle explicitly, either by using a try-catch block or declaring the exception in the method signature using the throws keyword.
    - Examples of checked exceptions include:
      * IOException: Occurs when there are input/output problems (e.g., when reading or writing a file).
      * SQLException: Happens when there's an issue with database access.
      * ClassNotFoundException: Raised when the Java runtime can't find a class at runtime.

Unchecked Exceptions (Runtime Exceptions):

* These are exceptions that are not required to be declared or caught by the program. They generally indicate programming bugs (e.g., logic errors, improper input).
* Unchecked exceptions are subclasses of RuntimeException.
* Examples of unchecked exceptions include:
  + NullPointerException: Occurs when trying to access a method or field of a null object.
  + ArrayIndexOutOfBoundsException: Happens when trying to access an array with an invalid index.
  + ArithmeticException: Occurs, for example, when dividing by zero.

Differences Between Errors and Exceptions:

| Aspect | Errors | Exceptions |
| --- | --- | --- |
| Cause | Typically caused by system-level problems, like memory overflow, JVM issues. | Caused by programmatic issues (e.g., bad input, logic errors, external problems). |
| Recoverability | Generally not recoverable by the program. | Often recoverable, especially with proper exception handling. |
| Handling | Not typically handled by code. | Can be handled using try-catch blocks or declared using throws. |
| Type | Subclasses of Error (e.g., OutOfMemoryError). | Subclasses of Exception (e.g., IOException, NullPointerException). |
| Example | OutOfMemoryError, StackOverflowError | IOException, NullPointerException, ArithmeticException. |

Summary:

* Errors are typically fatal and indicate problems with the system, which usually cannot be recovered from.
* Exceptions represent issues that arise during program execution and can often be handled or recovered from using proper exception handling mechanisms.
  + Checked Exceptions: Must be either caught or declared.
  + Unchecked Exceptions (Runtime Exceptions): Can be avoided through proper code design but are not required to be explicitly handled.

--------------------------------------------------------------------------------------------------------------

Q). What is thread in Java?

In Java, a thread is a lightweight unit of execution within a program. It represents a single path of execution in a program. Java allows concurrent execution of multiple threads within a single program, making it possible to perform several tasks simultaneously. Threads are essential for multitasking and are commonly used to improve the performance of applications by executing multiple operations in parallel.

Key Concepts of Threads in Java:

1. Thread:
   * A thread in Java is an independent path of execution in a program. It is the smallest unit of a CPU's scheduling capability and can run concurrently with other threads.
2. Multithreading:
   * Multithreading refers to the ability of a CPU to provide multiple threads of execution concurrently. Java supports multithreading, allowing programs to execute multiple threads simultaneously.
   * Multithreading is particularly useful for applications that require concurrent tasks.

There are two main ways to create and start a thread in Java:

1. By Extending the Thread Class

2. By Implementing the Runnable Interface

Thread States:

A thread can be in one of the following states during its lifecycle:

1. New: The thread is created but not yet started (i.e., the start() method has not been called).
2. Runnable: The thread is ready to run and is waiting for the CPU to schedule it.
3. Blocked: The thread is blocked, usually waiting for a resource (e.g., file, data).
4. Waiting: The thread is waiting indefinitely for another thread to perform a specific action (e.g., join()).
5. Timed Waiting: The thread is waiting for a specific period of time (e.g., sleep()).
6. Terminated: The thread has completed its execution (either normally or by an exception).

----------------------------------------------------------------------------------------------------------------------

Q). What is a Tree map?

In Java, a Tree Map is a data structure that implements the Sorted Map interface. It stores key-value pairs in a sorted order based on the natural ordering of the keys or a custom comparator.

Key Characteristics:

* Sorted Order: The keys in a Tree Map are always sorted in ascending order (by default) based on their natural ordering or a custom comparator. This means you can efficiently retrieve elements in sorted order.
* Unique Keys: Each key in a Tree Map must be unique. Duplicate keys are not allowed.
* Self-Balancing Tree: Tree Map internally uses a Red-Black tree, which is a self-balancing binary search tree. This ensures that operations like insertion, deletion, and retrieval have a time complexity of O(log n) in the average case.
* Natural Ordering: By default, Tree Map uses the natural ordering of the keys. For example, if the keys are integers, they will be sorted in ascending numerical order. If the keys are strings, they will be sorted in lexicographical order.
* Custom Comparator: You can provide a custom Comparator object to Tree Map to specify a different sorting order for the keys.

Advantages:

* Sorted Order: Provides efficient access to elements in sorted order.
* Fast Lookups: Efficient for searching, insertion, and deletion operations.
* Flexibility: Allows for custom sorting order using a Comparator.

Disadvantages:

* Slower than HashMap: Generally slower than HashMap for operations that don't rely on sorted order.
* Higher Memory Overhead: May require slightly more memory than HashMap due to the overhead of maintaining the self-balancing tree structure.

Use Cases:

* When you need to access data in sorted order.
* When you frequently need to find the minimum or maximum key.
* When you want to use a custom sorting order for the keys.

### **Comparison: TreeMap vs HashMap**

| Feature | **TreeMap** | **HashMap** |
| --- | --- | --- |
| **Sorting** | Stores keys in sorted order (natural or custom comparator). | Does not guarantee any order. |
| **Performance** | Slower operations due to the need to maintain order (O(log n)). | Faster operations (O(1) average time). |
| **Null Keys** | Does not allow null keys. | Allows one null key. |
| **Thread Safety** | Not synchronized. | Not synchronized. |
| **Used for** | When sorted order of keys is required. | When you need fast lookup and insertion without order. |

---------------------------------------------------------------------------------------------------------------------

Q). What is the difference between String and StringBuilder?

In Java, String and StringBuilder are both used to handle text data. However, there are some fundamental differences between them, particularly in how they manage memory and performance.

1. Immutability vs Mutability

String:

Immutable: A String object in Java is immutable, meaning once it is created, its value cannot be changed. Any operation that modifies a String (like concatenation, replacing characters, etc.) will create a new String object in memory, leaving the original one unchanged.

StringBuilder:

Mutable: A StringBuilder object is mutable, meaning its content can be modified without creating new objects. This makes StringBuilder much more efficient when performing repeated modifications to strings (like concatenation).

2. Performance

String:

Because String is immutable, concatenating strings using the + operator in a loop repeatedly creates new String objects, which can be inefficient, especially for large-scale string manipulation.

* + Example: In a loop, string concatenation using String can cause a lot of temporary object creation, resulting in poor performance.

StringBuilder:

Since StringBuilder is mutable, it is much more efficient for repeated modifications like concatenation. It internally uses a char array and modifies the content of the array without creating new objects. This results in much better performance when working with strings in loops.

3. Memory Usage

String:

* + Since String objects are immutable, every modification creates a new object, which increases memory consumption, especially if you are performing many modifications or concatenations.

StringBuilder:

* + StringBuilder reuses the internal buffer to store the string. When more capacity is required, it doubles the size of the internal array. It is generally more memory-efficient in situations with frequent string modifications.

4. Thread-Safety

String:

String objects are thread-safe because they are immutable. Since the content of a String cannot change after it is created, there are no issues with concurrent access or modification.

StringBuilder:

StringBuilder is not thread safe. Since its content can be modified, multiple threads accessing a StringBuilder object concurrently could cause data inconsistency or corruption. If thread-safety is required, you should use String Buffer (a synchronized version of StringBuilder) or manage synchronization manually.

5. Usage Scenarios

String:

Use String when the string content does not change after creation, or if it changes infrequently.

Suitable for working with fixed strings or when dealing with string literals.

Example:

Storing a constant value like "Hello World".

When passing strings between methods or APIs that expect String.

StringBuilder:

Use StringBuilder when you need to modify the string content frequently or perform many concatenations (like in loops or dynamic string construction).

Ideal for scenarios where performance is critical, and string modifications are needed.

Example:

Building a large string in a loop or when parsing data dynamically.

6. Thread-Safety Consideration

String:

Since String is immutable, it is inherently thread-safe and can be safely used across multiple threads without synchronization.

StringBuilder:

StringBuilder is not thread-safe, meaning if you use it in a multi-threaded environment where multiple threads may access or modify the same instance concurrently, you may encounter issues unless you manage synchronization yourself.

7. Capacity and Growth

String:

A String has no concept of "capacity" in the same way a StringBuilder does. It simply represents a sequence of characters.

StringBuilder:

A StringBuilder starts with an initial capacity (usually 16 characters by default) and grows dynamically when more space is needed. When the internal buffer is full, it doubles the buffer size, which helps avoid frequent reallocation.

Summary of Key Differences

| Feature | String | StringBuilder |
| --- | --- | --- |
| Immutability | Immutable (creates new object on modification) | Mutable (modifies the same object) |
| Performance | Slower for repeated modifications | Faster for repeated modifications |
| Memory Efficiency | Less memory-efficient due to object creation | More memory-efficient as it reuses the buffer |
| Thread Safety | Thread-safe | Not thread-safe (use StringBuffer for thread-safety) |
| Use Case | Suitable for constant or rarely modified strings | Suitable for dynamically built strings, frequent concatenation |
| Capacity Growth | Does not have a capacity, fixed length | Grows dynamically as required, with automatic resizing |

--------------------------------------------------------------------------------------------------------------

* What is the difference between String and StringBuilder?

| Feature | String | StringBuilder |
| --- | --- | --- |
| Mutability | Immutable (cannot be modified) | Mutable (can be modified) |
| Performance | Slower for frequent modifications | Faster for frequent modifications |
| Memory Efficiency | Less memory-efficient (creates new objects on modification) | More memory-efficient (modifies the same object) |
| Thread Safety | Thread-safe | Not thread-safe |
| Use Case | When text is constant or rarely changes | When frequent modifications are needed |
| Constructor | String constructor, or literal | StringBuilder constructor |
| Methods | Methods like concat(), substring() | Methods like append(), delete(), reverse() |

Conclusion:

* Use String when:
  + The text is immutable and does not need frequent modification.
  + The string value is constant, such as names, constants, or error messages.
  + You want thread-safety because strings are immutable and can be safely shared across threads.
* Use StringBuilder when:
  + You need to perform frequent modifications (like appending, inserting, or deleting characters).
  + You're dealing with a large number of string operations in loops or performance-sensitive code.
  + Memory efficiency is important because StringBuilder avoids creating new objects for each modification.

In summary, String is ideal for immutable text, while StringBuilder is the better choice for scenarios requiring efficient and mutable string manipulation.

* Difference between equals and ==

In Java, both equals() and == are used to compare objects, but they serve different purposes and behave in different ways. Here's a detailed explanation of the difference between equals() and ==:

1. == (Reference Comparison):

* Purpose: The == operator is used to compare references (memory addresses) of two objects, not their actual content.
* Behavior:
  + For primitive types (like int, char, float, etc.), == compares their values.
  + For objects, == checks whether two references point to the same object in memory (i.e., whether they are the same instance).
* Use Case:
  + == is used when you want to check if two references are pointing to the exact same object (i.e., whether they refer to the same memory location).
* Example:

java

Copy code

String str1 = new String("Hello");

String str2 = new String("Hello");

System.out.println(str1 == str2); // false, because str1 and str2 are different objects in memory

In the example above, str1 and str2 are two different String objects, even though their contents are the same. Therefore, str1 == str2 returns false.

2. equals() (Content Comparison):

* Purpose: The equals() method is used to compare the contents or values of two objects, not their references.
* Behavior:
  + It is defined in the Object class and is overridden in many classes (like String, Integer, List, etc.) to compare the values of objects.
  + By default, the equals() method in the Object class behaves the same as == (it checks reference equality), but in classes like String, it compares the content of the strings.
* Use Case:
  + equals() is used when you want to check if two objects have the same content, even if they are not the same instance.
* Example:

String str1 = new String("Hello");

String str2 = new String("Hello");

System.out.println(str1.equals(str2)); // true, because the contents of str1 and str2 are the same

In this case, str1.equals(str2) returns true because the contents of both strings ("Hello") are the same, even though they are different objects in memory.

Key Differences:

| Aspect | == | equals() |
| --- | --- | --- |
| Comparison Type | Compares references (memory addresses). | Compares contents (values). |
| Default Behavior | Checks if two references point to the same object in memory. | By default, checks if two references point to the same object (like ==), but many classes override it to compare actual content. |
| Used for Primitives? | Yes, compares values for primitive types (e.g., int, char). | No, cannot be used for primitive types. |
| Overridden in Classes? | No, == cannot be overridden. | Yes, many classes override it (e.g., String, List, Integer). |
| Example | str1 == str2 checks if both references point to the same object. | str1.equals(str2) checks if both objects have the same content. |

Summary:

* Use == when you want to check if two references point to the same object.
* Use equals() when you want to check if the content of two objects is the same.

In most cases when dealing with objects, especially String and other custom objects, you should use equals() to compare their content rather than ==. However, if you are comparing object references or working with primitive types, == is the correct operator.

* What is string pool?

What is the String Pool in Java?

In Java, the String Pool (also known as the String Constant Pool or String Literal Pool) is a special memory area in the heap where String objects are stored. The String Pool is designed to optimize memory usage and improve performance by reusing string literals.

Java strings are immutable (i.e., their values cannot be changed once they are created). The String Pool helps Java efficiently manage memory by storing unique string literals only once and reusing them throughout the program.

How the String Pool Works:

1. String Literals:
   * When you create a String literal (i.e., a string defined in your code directly, such as "hello"), Java first checks the String Pool to see if that string already exists.
   * If the string exists in the pool, Java will use the existing string reference. If it does not exist, the string will be added to the pool.
   * The String Pool ensures that only one copy of each string literal is created, saving memory and improving performance.
2. Example:

String str1 = "hello";

String str2 = "hello";

System.out.println(str1 == str2); // true, because both refer to the same object in the String Pool

In this example, both str1 and str2 will reference the same object from the String Pool, and the comparison str1 == str2 will return true.

1. String Objects Created with new:
   * When you create a String object using the new keyword (e.g., new String("hello")), Java creates a new object in memory, regardless of whether the string literal already exists in the pool.
   * However, Java will still add the string literal to the pool if it is not already present, but the object created with new is a separate instance from the one in the pool.
   * Example:

String str1 = new String("hello");

String str2 = "hello";

System.out.println(str1 == str2); // false, because str1 refers to a new object, and str2 refers to the string in the pool

1. Here, str1 will not refer to the string from the pool but will point to a new object created on the heap.

Benefits of the String Pool:

1. Memory Efficiency:
   * Since string literals are stored only once in the pool, memory usage is reduced when the same string is used multiple times in the program.
2. Performance Improvement:
   * String comparison using == (reference comparison) is much faster than using .equals() (content comparison), because with string literals, all strings with the same value in the pool refer to the same object.
3. Garbage Collection:
   * String literals in the pool are managed by the JVM and are typically not garbage collected unless the JVM decides to unload the class (though this rarely happens).

Important Points:

* The String Pool only stores string literals (i.e., strings defined in source code as "value"), not strings created using the new keyword.
* If you use new String("hello"), a new object is created in the heap, and that object is not stored in the String Pool.
* String literals are stored in the String Pool at compile-time, while strings created dynamically using new are not added to the pool by default.

Example of String Pool Behavior:

1. String literals:

String str1 = "hello";

String str2 = "hello";

System.out.println(str1 == str2); // true, both point to the same object in the String Pool

1. Using new keyword:

java

Copy code

String str1 = new String("hello");

String str2 = "hello";

System.out.println(str1 == str2); // false, str1 points to a new object, and str2 points to the pool

String Pool vs Heap:

* The String Pool is part of the heap memory, but it’s a specialized area dedicated to managing string literals.
* Objects created with new (like new String("hello")) are stored in the heap memory (but not in the String Pool).

Conclusion:

The String Pool is a performance optimization technique in Java that ensures that only one copy of each String literal exists in memory, which reduces memory usage and increases efficiency. By reusing string literals, Java minimizes the need to create multiple instances of the same string and makes string comparison faster with the == operator.

* What are wrapper classes?

In Java, Wrapper classes are used to wrap primitive data types (like int, char, double, etc.) into objects. Since Java is object-oriented, it requires objects for many operations, such as using collections (like ArrayList, HashMap, etc.), which can only store objects, not primitives. Wrapper classes allow primitive data types to behave like objects when needed.

Java provides a wrapper class for each primitive data type, and these classes are part of the java.lang package.

List of Wrapper Classes for Primitive Types:

| Primitive Type | Wrapper Class |
| --- | --- |
| boolean | Boolean |
| byte | Byte |
| char | Character |
| short | Short |
| int | Integer |
| long | Long |
| float | Float |
| double | Double |

Purpose of Wrapper Classes:

1. Object Representation of Primitives:
   * Java uses objects for many operations, such as storing data in collections like ArrayList, HashMap, and others, which require objects, not primitive types. Wrapper classes enable the use of primitive values as objects in these contexts.
2. Utility Methods:
   * Wrapper classes provide utility methods that allow converting between types, parsing strings, or performing operations that primitive types can't do.
   * For example, the Integer class provides methods like parseInt() and toString().
3. Autoboxing and Unboxing:
   * Java automatically converts (or "boxes") a primitive type into its corresponding wrapper class object (autoboxing), and vice versa (unboxing). This feature was introduced in Java 5.

Example of Autoboxing and Unboxing:

int num = 10;

Integer intObject = num; // Autoboxing (primitive int to Integer object)

int newNum = intObject; // Unboxing (Integer object to primitive int)

Wrapper Classes and Their Methods:

Each wrapper class has methods that can be used for various purposes:

1. Conversion Methods:
   * intValue(), floatValue(), doubleValue(), etc., allow you to convert the wrapper object back to its primitive type.

Integer num = new Integer(10);

int primitiveNum = num.intValue(); // Converts Integer to int

1. Parsing Strings:
   * Wrapper classes provide methods to convert strings to their corresponding primitive types.

int parsedInt = Integer.parseInt("123"); // Converts a String to a primitive int

double parsedDouble = Double.parseDouble("3.14"); // Converts String to primitive double

1. String Representation:
   * toString() method can be used to convert the wrapper object into its string form.

Integer num = new Integer(10);

String str = num.toString(); // Converts Integer object to String

1. Constants:
   * Each wrapper class has constants like MIN\_VALUE and MAX\_VALUE that represent the extreme limits for each primitive type.

System.out.println(Integer.MAX\_VALUE); // Output: 2147483647

System.out.println(Double.MIN\_VALUE); // Output: 4.9e-324

Examples of Using Wrapper Classes:

1. Autoboxing:

List<Integer> intList = new ArrayList<>();

intList.add(5); // Autoboxing, primitive int is converted to Integer

1. Unboxing:

Integer integerObject = new Integer(10);

int primitiveInt = integerObject; // Unboxing, Integer is converted to primitive int

1. Utility Methods:

String str = "123";

int num = Integer.parseInt(str); // Convert String to int

System.out.println(num); // Output: 123

1. Comparison:

Integer num1 = new Integer(10);

Integer num2 = new Integer(10);

System.out.println(num1 == num2); // false, because different objects

System.out.println(num1.equals(num2)); // true, because same value

Autoboxing and Unboxing in Detail:

* Autoboxing: Java automatically converts primitive types to wrapper objects when needed.

Integer intObj = 10; // Autoboxing of primitive int to Integer object

* Unboxing: Java automatically converts wrapper objects back to primitive types when needed.

int primitiveValue = intObj; // Unboxing of Integer object to primitive int

Why Use Wrapper Classes?

1. Collection Framework:
   * The Java Collection Framework (e.g., List, Set, Map) can only store objects, not primitive types. Therefore, wrapper classes are used to store primitive values in collections.
2. Utility Methods:
   * Wrapper classes provide several utility methods like parseInt(), valueOf(), and toString(), making it easier to manipulate and convert data types.
3. Null Handling:
   * Primitive types cannot hold null, but wrapper classes can. This is useful in situations like database handling, where you need to represent an unknown or missing value as null.
4. Working with Generic Types:
   * Java generics require objects, and since primitive types cannot be used with generics, wrapper classes are used instead.

List<Integer> numbers = new ArrayList<>(); // Only Integer (not int) can be used in generics

Summary of Wrapper Classes:

* Wrapper classes allow primitive types to be used as objects.
* They provide methods for conversion between primitive types and strings.
* They enable primitives to be used in collections, generics, and null-handling situations.
* Autoboxing and Unboxing simplify the conversion between primitive types and wrapper objects.
* Java provides one wrapper class for each primitive type (e.g., Integer for int, Character for char, Double for double, etc.).

Wrapper classes play an important role in Java, especially when working with object-oriented features, collections, and APIs that require object types.

* Types of exception

In Java, exceptions are classified into two main types:

1. Checked Exceptions
2. Unchecked Exceptions

1. Checked Exceptions

* These are exceptions that are checked at compile-time by the Java compiler. The compiler ensures that these exceptions are either caught using a try-catch block or declared in the method signature using the throws keyword.
* If a checked exception is not properly handled, the compiler will generate an error.

Characteristics:

* Must be either caught or declared in the method signature.
* Typically represent conditions that are external to the application, like file handling, network issues, etc.

Examples of Checked Exceptions:

* IOException: Thrown when an I/O operation fails, such as reading or writing a file.
* SQLException: Thrown when there is a database access error.
* FileNotFoundException: Thrown when a file cannot be found.
* ClassNotFoundException: Thrown when a class cannot be found by the class loader.

2. Unchecked Exceptions

* These are exceptions that are not checked at compile-time. They are also called runtime exceptions.
* Unchecked exceptions usually represent programming errors, such as accessing invalid array indices or dividing by zero.
* These exceptions extend the RuntimeException class, which in turn extends the Exception class.
* The programmer is not required to catch or declare these exceptions, although they can be caught.

Characteristics:

* Not required to be caught or declared in the method signature.
* Typically caused by programming errors or logic mistakes that the programmer should correct.
* Runtime exceptions are unchecked, and errors are usually a result of invalid operations.

Examples of Unchecked Exceptions:

* NullPointerException: Thrown when trying to access a method or field on a null object reference.
* ArrayIndexOutOfBoundsException: Thrown when trying to access an array index that is out of bounds.
* ArithmeticException: Thrown when an arithmetic operation (like division by zero) is performed.
* ClassCastException: Thrown when an object cannot be cast to a different class.
* NumberFormatException: Thrown when trying to convert a string to a number but the string doesn't have a valid number format.

3. Error

* Errors are not exceptions but represent severe problems that are beyond the control of the application.
* These are usually generated by the JVM and are not recoverable. Most errors indicate serious issues such as out of memory, stack overflow, or JVM internal failures.
* Errors are not handled using try-catch blocks.

Examples of Errors:

* OutOfMemoryError: Thrown when the Java Virtual Machine (JVM) runs out of memory.
* StackOverflowError: Thrown when the stack overflows, usually due to deep recursion.
* AssertionError: Thrown when an assertion fails (used in debugging).

Summary of Exception Types:

| Type | Checked | Unchecked | Error |
| --- | --- | --- | --- |
| Inherit From | Exception (but not RuntimeException) | RuntimeException | Error |
| Checked by Compiler | Yes | No | No |
| Examples | IOException, SQLException | NullPointerException, ArithmeticException | OutOfMemoryError, StackOverflowError |
| Usage | Need to be caught or declared | Can be ignored or caught optionally | Usually not handled in code, represents critical errors |

Conclusion:

* Checked exceptions are typically external errors that a program can handle (e.g., I/O or database issues).
* Unchecked exceptions usually represent programming bugs that should be fixed by the developer (e.g., NullPointerException or ArrayIndexOutOfBoundsException).
* Errors represent system-level issues that cannot be handled or recovered from, such as memory errors or JVM internal failures.

Understanding and handling exceptions correctly allows you to write more robust and error-resilient Java programs.

* What is use of spring?

Spring is a comprehensive framework for building enterprise-level Java applications. It is one of the most widely used frameworks in the Java ecosystem due to its flexibility, simplicity, and wide range of functionalities. Spring is designed to make Java development easier, more efficient, and more robust.

Spring provides a wide array of features, including dependency injection, aspect-oriented programming (AOP), transaction management, web applications, and more. It's used to build scalable, maintainable, and flexible applications.

Main Uses of Spring Framework

1. Dependency Injection (DI)
   * Spring provides an Inversion of Control (IoC) container, which facilitates Dependency Injection. This means that Spring manages the instantiation and wiring of objects and their dependencies.
   * DI allows for loose coupling between components, which simplifies testing, configuration, and maintenance.
   * Example: You can define dependencies in XML or using annotations and Spring will inject them at runtime.
2. Aspect-Oriented Programming (AOP)
   * AOP allows for separation of concerns by allowing developers to modularize cross-cutting concerns (e.g., logging, transaction management, security).
   * It helps in adding functionality to existing code without modifying the original code, which is useful for tasks like logging, authentication, and transaction management.
   * Example: You can define an aspect that logs method execution without changing the method itself.
3. Spring MVC (Model-View-Controller)
   * Spring MVC is a robust framework for building web applications. It implements the MVC pattern, which separates concerns between the model (data), view (user interface), and controller (business logic).
   * It provides support for various view technologies, such as JSP, Thymeleaf, or other templating engines.
   * Spring MVC supports RESTful web services, which means it can be used to build modern, REST-based APIs for web and mobile applications.
4. Transaction Management
   * Spring provides a consistent transaction management interface for managing database transactions in Java applications.
   * It integrates with Java’s JTA, JDBC, Hibernate, and other ORM frameworks, allowing for declarative or programmatic transaction management.
   * This ensures that transactions are managed in a way that guarantees consistency, even in the event of failure.
5. Spring Boot
   * Spring Boot is an extension of the Spring framework that simplifies the configuration and deployment of Spring-based applications.
   * It is designed to create standalone applications with minimal configurations.
   * With Spring Boot, developers can create production-ready applications that run on embedded servers (like Tomcat or Jetty), making it easier to deploy microservices and cloud-based applications.
   * Example: Spring Boot provides out-of-the-box configurations and starter templates to speed up the development process.
6. Security Framework (Spring Security)
   * Spring Security is a powerful and customizable authentication and access-control framework.
   * It supports features like authentication, authorization, secure HTTP headers, session management, and more.
   * Spring Security can be easily integrated with Spring applications to secure APIs, web applications, and services.
   * Example: You can easily integrate user login, role-based access control, and OAuth authentication in a Spring-based application.
7. Spring Data and Spring ORM
   * Spring simplifies database access and integration with Spring Data. It provides JPA (Java Persistence API), JDBC (Java Database Connectivity), and Hibernate integration.
   * Spring Data offers repository-based abstractions that allow developers to easily interact with databases without writing a lot of boilerplate code.
   * Example: You can use Spring Data repositories to automatically implement common database operations without manually writing SQL or HQL queries.
8. Spring Cloud for Microservices
   * Spring Cloud provides tools for building distributed systems and microservices.
   * It offers solutions for service discovery (e.g., Netflix Eureka), load balancing (e.g., Netflix Ribbon), and distributed configuration (e.g., Spring Cloud Config Server).
   * Example: You can build scalable microservices applications with features like service discovery, messaging, and circuit breakers using Spring Cloud.
9. Spring Batch
   * Spring Batch is a framework for developing batch processing applications, commonly used for handling large volumes of data processing.
   * It provides features for reading, processing, and writing data in a reliable and scalable manner.
   * Example: You can process large datasets in chunks (e.g., batch import/export) and manage task scheduling and error handling efficiently.
10. Spring Integration
    * Spring Integration provides an extension of the Spring framework for enterprise integration. It allows building messaging-based systems and integrates various systems using channels, transformers, filters, etc.
    * Example: It can be used for integrating legacy systems with modern applications, messaging systems like JMS, or external services.

Benefits of Using Spring Framework

1. Simplifies Development:
   * Spring's dependency injection helps decouple the components of an application, making it easier to develop, test, and maintain.
   * It reduces boilerplate code and configuration, especially with Spring Boot, which makes application development faster.
2. Modularity:
   * Spring is modular, so developers can use only the parts of the framework they need. For example, you can use Spring’s IoC container or only the Spring MVC module for building web apps.
3. Cross-Cutting Concerns:
   * Spring’s AOP capabilities allow developers to handle cross-cutting concerns like logging, security, and transactions separately, making the code cleaner and more maintainable.
4. Extensive Ecosystem:
   * Spring has a vast ecosystem of projects, including Spring Security, Spring Batch, Spring Cloud, Spring Data, and more. This allows developers to easily integrate with a wide range of third-party systems and technologies.
5. Testability:
   * Spring's dependency injection and modularity make it easy to write unit tests and integration tests. Spring provides tools like JUnit and Mockito integration to test components in isolation.
6. Open Source and Large Community:
   * Spring is open-source and has an active community of developers and contributors. It has a wide range of resources, including documentation, tutorials, and forums for learning and support.

When to Use Spring?

* Enterprise Applications: When building large-scale, maintainable, and scalable enterprise applications, especially when there is a need for integration with other systems, databases, or messaging systems.
* Web Applications: When developing web applications with RESTful APIs, MVC architecture, or when you need microservices and distributed architectures.
* Microservices: When you want to design, deploy, and manage microservices-based applications, especially with the help of Spring Cloud and Spring Boot.
* Security: If your application requires secure authentication, authorization, and user management, Spring Security makes it easy to implement these features.
* Batch Processing: If you need to handle large volumes of data processing, such as importing/exporting data or running periodic jobs, Spring Batch provides a comprehensive solution.

### **Conclusion**

Spring is a powerful and flexible framework that simplifies Java application development, especially for building enterprise-level applications. It covers a wide range of concerns, from dependency injection to transaction management, security, microservices, and more. The modular nature of Spring means you can pick and choose only the features you need, making it a versatile and essential tool in Java development.

* What is loC?

LoC stands for Line of Code. It is a software metric used to measure the size of a program by counting the number of lines in the source code. LoC can be used to gauge various aspects of software development, such as complexity, effort, and productivity.

However, it's important to note that Line of Code is not always the best indicator of software quality or functionality, as it does not account for the complexity, readability, or performance of the code.

Understanding LoC:

* Line of Code typically counts the total number of lines in a program, including both actual code lines and sometimes blank lines or comments, depending on the specific metric definitions.
* In some contexts, only non-empty, non-comment lines are counted, while in others, all lines including comments and blank spaces are considered.

Key Uses of LoC (Line of Code):

1. Measuring Program Size:
   * LoC can be used to determine the size of the codebase. For example, a project with 10,000 lines of code can be considered larger than a project with 1,000 lines of code.
2. Estimating Development Effort:
   * LoC can serve as a rough indicator of the development effort required for a project. Larger codebases might take more time and resources to develop, maintain, and debug, although this is not always the case.
3. Productivity Measurement:
   * Developers or teams may use LoC to track how much code they are writing. However, this metric is controversial since more code does not necessarily mean better or more efficient software.
4. Code Quality:
   * A high LoC count might indicate a more feature-rich application, but it could also reflect poor design choices, such as unnecessary duplication or overly verbose code.
5. Maintenance Effort:
   * Larger codebases (i.e., higher LoC) often require more effort to maintain, test, and debug, especially if the code is poorly structured.
6. Comparison Between Projects:
   * LoC is sometimes used to compare the size of different software projects. A project with more lines of code might be expected to be more complex or offer more features than a smaller one.

Types of LoC Measurements:

1. Physical LoC:
   * Counts every line in the source code, including blank lines and comments.
2. Logical LoC:
   * Counts only lines that contain actual executable code, excluding blank lines and comments.
3. Executable LoC:
   * Includes only lines that produce an effect when the program is run (for example, a line that executes a function or a variable assignment).

Example:

Here is a simple piece of code to illustrate the concept of LoC:

public class LoCExample { // Line 1

public static void main(String[] args) { // Line 2

int sum = 0; // Line 3

for (int i = 1; i <= 10; i++) { // Line 4

sum += i; // Line 5

} // Line 6

System.out.println("Sum: " + sum); // Line 7

} // Line 8

} // Line 9

* Total physical lines of code (LoC): 9 lines
* Total logical lines of code: 6 lines (excluding blank lines and comments)

Limitations of LoC as a Metric:

1. Doesn't Reflect Quality:
   * LoC does not reflect the quality of the code, such as whether it’s well-structured, easy to maintain, or free of bugs. Writing fewer lines of high-quality code is often more valuable than writing more lines of poor-quality code.
2. Encourages Quantity Over Quality:
   * Developers might focus on increasing LoC to appear more productive, leading to verbose or unnecessary code.
3. Does Not Measure Complexity:
   * LoC does not measure the complexity of an algorithm or the system as a whole. Two programs with the same LoC could have vastly different levels of complexity.
4. Maintenance Challenge:
   * A larger codebase can make maintaining the software more difficult. More lines of code often imply a higher potential for bugs and longer debugging or testing times.

Conclusion:

Line of Code (LoC) is a simple metric to measure the size of a program by counting the number of lines written in the source code. While it can be useful for certain project management tasks like estimating development time or comparing projects, it should not be used as the sole measure of code quality or productivity. It's important to consider other factors like code clarity, efficiency, maintainability, and functionality alongside LoC for a more comprehensive understanding of software development progress.

* What is microservices?

Microservices

* Definition: Microservices are a software architectural style where a single application is broken down into a collection of small, independent services.
* Key Characteristics:
  + Small, Independent Services: Each service is responsible for a specific business capability and runs as an independent process.
  + Loosely Coupled: Services communicate with each other through lightweight mechanisms like APIs (typically RESTful APIs).
  + Decentralized Governance: Each service can be developed, deployed, and scaled independently.
  + Business Capabilities: Services are organized around business capabilities, making them more agile and easier to understand.
  + Data Decoupling: Each service typically owns its own data.

Benefits of Microservices:

* Improved Agility: Enables faster development and deployment cycles due to the independent nature of services.
* Scalability: Allows for fine-grained scaling of individual services based on demand.
* Fault Isolation: If one service fails, it doesn't necessarily bring down the entire application.
* Technology Heterogeneity: Different services can use different technologies and programming languages.
* Continuous Delivery: Enables continuous integration and continuous delivery practices.

Challenges of Microservices:

* Increased Complexity: Managing a large number of services can be complex.
* Distributed System Challenges: Issues like network latency, service discovery, and data consistency need to be addressed.
* Testing and Debugging: Testing and debugging distributed systems can be more challenging.

Examples:

* A social media platform could be broken down into microservices for user management, messaging, notifications, and content recommendations.
* An e-commerce platform could have microservices for product catalog, order management, payment processing, and inventory management.

In summary:

Microservices offer a modern approach to building complex applications by breaking them down into smaller, manageable, and independently deployable services. While they bring significant benefits, they also introduce new challenges that need to be carefully considered and addressed.

* What is spring cloud?

Spring Cloud is an umbrella project that provides a comprehensive set of tools and libraries for building cloud-native applications on top of the Spring Platform. It aims to simplify the development of complex distributed systems, such as microservices architectures.

Key Features and Components:

* Service Discovery: Enables services to automatically discover and locate each other within the network.
* Configuration Management: Provides tools for managing and distributing configuration across different environments.
* Circuit Breaker: Helps to handle service failures gracefully by implementing circuit breaker patterns.
* Load Balancing: Distributes traffic across multiple instances of a service to improve performance and fault tolerance.
* API Gateway: Provides a single entry point for clients to access microservices, handling routing, authentication, and security.
* Distributed Tracing: Enables tracing requests across multiple services to identify performance bottlenecks and troubleshoot issues.

Benefits of using Spring Cloud:

* Simplified Development: Provides pre-built solutions for common challenges in cloud-native development, reducing development time and effort.
* Improved Resilience: Helps build more resilient applications that can handle failures and disruptions gracefully.
* Enhanced Scalability: Enables easy scaling of individual services based on demand.
* Improved Developer Productivity: Provides a productive and efficient development experience with features like auto-configuration and convention over configuration.

Key Components of Spring Cloud:

* Spring Cloud Netflix: Provides integration with Netflix OSS components like Eureka (service discovery), Hystrix (circuit breaker), and Ribbon (load balancing).
* Spring Cloud Config: Enables centralized configuration management for microservices.
* Spring Cloud Gateway: Provides a powerful API gateway for routing, filtering, and security.
* Spring Cloud Sleuth: Enables distributed tracing for microservices.

In summary:

Spring Cloud is a valuable toolkit for developers building microservices architectures on the Spring Platform. It provides a set of tools and abstractions that simplify common challenges and enable the development of robust, scalable, and resilient distributed systems.

* What is entity manager?

EntityManager is the central interface in JPA for managing entities, performing CRUD operations, and interacting with the database. It abstracts much of the complexity involved in dealing with the underlying database and provides a powerful set of methods to interact with entities. By managing the persistence context, handling transactions, and allowing for querying via JPQL or native SQL, the EntityManager is essential for developers working with JPA in Java-based enterprise applications.

EntityManager is a key component in the Java Persistence API (JPA), which is part of the Java EE (Enterprise Edition) specification for managing relational data in Java applications. The EntityManager is responsible for managing the lifecycle of entities (Java objects mapped to database tables) and providing an interface for interacting with the persistence context. It allows developers to perform operations like creating, reading, updating, and deleting entities from a database.

In simpler terms, EntityManager is the main interface used for interacting with the database and performing CRUD (Create, Read, Update, Delete) operations in a Java-based persistence context, usually in combination with an ORM (Object-Relational Mapping) framework like Hibernate.

* Why we use Rest API over Soap.?

Both REST (Representational State Transfer) and SOAP (Simple Object Access Protocol) are popular approaches for building web services, but REST has become the preferred choice for modern web applications due to several reasons. Below is a comparison of both, highlighting why REST is often chosen over SOAP:

1. Simplicity

* REST:
  + REST is much simpler to use and more lightweight than SOAP. It is based on standard HTTP protocols (such as GET, POST, PUT, DELETE), making it easy to work with. It uses URL paths, and HTTP headers, and data is typically transmitted in lightweight formats like JSON or XML, with JSON being the most popular.
  + The design principles of REST focus on resources (data objects) and their interactions, making it intuitive and straightforward for developers to build and consume APIs.
* SOAP:
  + SOAP is more complex. It is a protocol with a strict set of rules and requires XML messages with additional overhead such as security features, and message routing, and often requires more setup. SOAP messages also need to be packaged in an XML envelope and may require a SOAP handler for request and response processing.
  + SOAP requires specific libraries to handle the protocol, leading to more complexity and configuration.

2. Flexibility (Data Formats)

* REST:
  + REST APIs can support multiple data formats, but the most common formats are JSON and XML. JSON is especially favored for its lightweight structure and ease of use with JavaScript, making it the go-to choice for modern web applications.
  + REST provides more flexibility as developers can choose their data formats based on application needs (e.g., XML, JSON, HTML, or even plain text).
* SOAP:
  + SOAP is limited to XML for messaging, which adds more complexity to the data processing and results in higher bandwidth usage due to its verbose nature. This makes SOAP more suited for certain enterprise systems but less ideal for lightweight or mobile applications.

3. Performance and Efficiency

* REST:
  + REST is lightweight and designed for fast performance. It works with standard HTTP methods and avoids the overhead of the XML-based message format used in SOAP.
  + Caching is built into RESTful systems because they typically rely on HTTP, which natively supports caching mechanisms. This enhances performance when retrieving the same data multiple times.
* SOAP:
  + SOAP is more heavyweight due to the XML envelope and the additional overhead in each message (envelopes, headers, etc.). It is slower than REST in terms of performance, especially when dealing with large amounts of data or requiring multiple calls.
  + SOAP does not natively support caching, which can lead to reduced performance when multiple requests for the same resources are needed.

4. Ease of Use

* REST:
  + REST is designed to be simple to understand and implement, especially for web and mobile applications. It uses standard HTTP verbs (GET, POST, PUT, DELETE) and leverages URL-based resource access.
  + REST APIs are more flexible in their design, allowing different ways to handle responses, errors, and status codes.
* SOAP:
  + SOAP requires more configuration and setup, and its rigid structure can make it more complex to work with, especially for developers not familiar with SOAP-specific tools and libraries.
  + SOAP APIs are typically more challenging to implement and use compared to RESTful APIs, requiring additional knowledge and configuration, such as handling the XML message structure, security protocols, and other standards.

5. Statelessness

* REST:
  + RESTful services are stateless, meaning that each request from a client to a server is independent. Each request contains all the information needed to understand and process the request. This stateless nature makes REST more scalable and fault-tolerant, as servers do not need to retain any session information between requests.
  + Statelessness also simplifies scalability, as the server does not need to remember the state of each client.
* SOAP:
  + SOAP can be stateful or stateless, but many SOAP web services are designed to be stateful, which can introduce complexity in managing session states and maintaining connections.

6. Security

* REST:
  + REST generally relies on HTTP security mechanisms, such as HTTPS for encryption. It is less standardized in terms of security features compared to SOAP.
  + You would typically handle authentication in REST via mechanisms like OAuth, API tokens, or basic authentication.
* SOAP:
  + SOAP has built-in security features as part of the WS-Security standard, which can handle things like encryption, digital signatures, and authentication. This makes SOAP more suitable for highly secure enterprise-level applications where complex security needs exist, such as in banking, payment systems, and telecommunication services.
  + However, these features introduce additional complexity in implementation.

7. Standardization

* REST:
  + REST is not a formal standard, but rather an architectural style. It provides flexibility in how APIs can be designed and is based on common web standards.
  + It is widely adopted in modern web development, particularly in mobile and cloud applications.
* SOAP:
  + SOAP is a protocol with strict standards defined by the W3C (World Wide Web Consortium). It provides a well-defined set of rules and practices, including message structure, error handling, and security, making it more suitable for complex, enterprise-level applications that require formal contracts.

8. Compatibility

* REST:
  + REST APIs are compatible with a wide range of clients, including web browsers, mobile devices, and other HTTP clients. They are generally used for web and mobile applications and are particularly well-suited for public-facing web services that are consumed by external clients.
* SOAP:
  + SOAP is more commonly used in enterprise-level applications that require more formal message contracts and heavy integration with existing enterprise systems (e.g., ERP or CRM systems).

9. Use Cases

* REST:
  + REST is the best choice for web-based applications, mobile applications, and cloud-based services, where simplicity, flexibility, and speed are important. Common use cases include:
    - Public APIs for third-party integrations (e.g., Google Maps, Twitter, etc.)
    - Mobile apps that require lightweight communication with a server.
    - REST APIs in microservices architecture.
* SOAP:
  + SOAP is more appropriate for enterprise applications that require formal contracts, such as banking systems, telecommunication services, and CRM/ERP systems where reliable messaging, security, and transaction management are crucial.

Summary of Why REST is Preferred Over SOAP:

| Feature | REST | SOAP |
| --- | --- | --- |
| Simplicity | Simple and easy to implement and use | More complex and requires additional setup |
| Data Format | Flexible (JSON, XML, HTML, etc.) | Only XML |
| Performance | Lightweight, faster performance (due to less overhead) | Slower due to XML overhead and additional features |
| Statelessness | Stateless (scalable and easy to manage) | Can be stateful or stateless |
| Security | Relies on HTTP security (e.g., HTTPS) | Built-in security via WS-Security |
| Standardization | No formal standard (more flexible) | Strict standards and protocols (e.g., WS-Security, WS-ReliableMessaging) |
| Use Case | Ideal for modern web, mobile, and cloud applications | Better suited for enterprise-level applications requiring formal contracts |

In conclusion, REST is preferred over SOAP for most modern web and mobile applications because it is easier to implement, more flexible, and more efficient, especially for lightweight services. However, SOAP still has a place in certain industries (e.g., finance, telecommunications) where strict security, reliability, and formal standards are required.

* What is starting point in java application

In a Java application, the starting point refers to the entry point where the program begins execution. For Java applications, this entry point is typically the main() method.

Main Method in Java

* The main() method is the starting point of any standalone Java application.
* It is a static method that serves as the entry point for the Java Virtual Machine (JVM) to start the execution of the program.
* It must be defined in a class, and when you run a Java application, the JVM looks for the main() method to begin the program execution.

Syntax of the main() method

public class MyApplication {

public static void main(String[] args) {

// Program logic goes here

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

}

}

Explanation:

* public: The method is public so that it can be accessed by the JVM from outside the class.
* static: The main() method is static, which means it belongs to the class itself and not to instances of the class. This allows the JVM to invoke it without creating an object of the class.
* void: The main() method returns void, meaning it does not return any value.
* String[] args: This is an array of strings that can be used to pass command-line arguments to the Java program. The args array holds values that the user inputs when starting the program from the command line (e.g., java MyApplication arg1 arg2).

Execution Flow

1. When you run a Java application, the JVM looks for the main() method in the class that is specified in the command.
2. It starts executing the code inside the main() method.
3. The application continues to run, executing the statements and calling methods, until the main() method completes or the program encounters an error.
4. Once the main() method finishes execution, the program terminates.

Key Points:

* In console-based Java applications, the main() method is the entry point for the program.
* In web applications (e.g., with servlets or Spring Boot), the program might not have a main() method directly. For example, in Spring Boot applications, the main() method is used to start the application, but the entry point might be more complex as it involves running a Spring context, initializing the web server, etc.

Example of Main Method in a Java Console Application:

public class MyApplication {

public static void main(String[] args) {

System.out.println("Java application started.");

// Create an instance of another class and call its methods

MyService service = new MyService();

service.performTask();

}

}

class MyService {

void performTask() {

System.out.println("Task is being performed.");

}

}

In Web Applications:

For web-based Java applications like those developed with Servlets or Spring Boot, the entry point is often handled by a container (like Tomcat or Jetty for Servlets or embedded servers in Spring Boot), and there is typically no need for a main() method to initiate the application logic directly.

* Spring Boot Application: In a Spring Boot application, the entry point is defined by the @SpringBootApplication annotation, and the main() method is used to launch the application.

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication

public class MySpringBootApplication {

public static void main(String[] args) {

SpringApplication.run(MySpringBootApplication.class, args);

}

}

In summary, the starting point in a typical Java application is the main() method, where the execution begins. For web applications or frameworks like Spring Boot, this may vary slightly, but in most cases, a main() method is still used to launch the application.

* What is difference between Compiler and Interpreter ?

Compiler

* Translation: Translates the entire source code into machine code (or an intermediate language like bytecode) in one go. This machine code can then be executed directly by the computer's processor.
* Execution: Once compiled, the program runs directly on the machine without the need for the compiler.
* Error Detection: Detects all errors in the entire program before execution.
* Performance: Generally faster execution speed because the machine code is optimized for the specific hardware.
* Examples: C, C++, Java (to bytecode)

Interpreter

* Translation: Translates and executes the source code line by line. It doesn't create an intermediate machine code file.
* Execution: Executes the code line by line, translating each line before executing it.
* Error Detection: Detects and reports errors as they occur during execution.
* Performance: Typically slower than compiled programs due to the overhead of interpreting code at runtime.
* Examples: Python, JavaScript, Ruby, PHP

Here's a table summarizing the key differences:

|  |  |  |
| --- | --- | --- |
| Feature | Compiler | Interpreter |
| Translation | Translates entire program at once | Translates and executes line by line |
| Output | Machine code (or bytecode) | No separate executable file |
| Execution | Faster execution | Slower execution |
| Error Detection | Detects all errors before execution | Detects errors during execution |
| Portability | Can be less portable as the generated machine code might be specific to the target architecture | Generally more portable across different platforms |

In essence:

* Compilers focus on optimizing for performance by translating the entire program upfront.
* Interpreters prioritize flexibility and ease of development by executing code line by line and providing immediate feedback.

Both compilers and interpreters have their own advantages and disadvantages, and the choice of which approach to use depends on the specific programming language and the requirements of the application.

* Interpreter based programming language?

Examples of Interpreter-Based Programming Languages:

1. Python:
   * Python code is interpreted line-by-line by the Python interpreter. This makes Python easy to use and modify but often slower compared to compiled languages.
2. JavaScript:
   * JavaScript is typically interpreted by the web browser (or a JavaScript engine like Node.js). Web browsers execute JavaScript code in real-time as a part of web page functionality.
3. Ruby:
   * Ruby is an interpreted language, where the Ruby interpreter translates and runs the code directly without needing a compilation phase.
4. PHP:
   * PHP is a server-side scripting language used for web development. PHP code is interpreted by the PHP interpreter on the server at runtime.
5. Perl:
   * Perl is also an interpreted language, meaning that the Perl interpreter reads and executes the code line-by-line.
6. Lisp:
   * Lisp is a family of programming languages that is often interpreted, where an interpreter processes and executes the code directly.
7. Bash/Shell scripting:
   * Shell scripting languages like Bash or Zsh are interpreted, where the shell reads and executes commands one by one.
8. MATLAB:
   * MATLAB is an interpreted language used for mathematical computing. MATLAB scripts are executed line-by-line by the MATLAB interpreter.

* What's singleton & its features?

The Singleton design pattern is a creational pattern that ensures a class has only one instance throughout the application's lifetime and provides a global point of access to that instance. It is commonly used in situations where you need to control access to a shared resource, such as a database connection, file system, or configuration settings, where only one instance is needed to handle all requests.

* Class and static class

In Java, classes and static classes are used for organizing code into reusable structures. However, static classes are a specific type of class that behaves differently from regular classes. Below is an explanation of the differences, characteristics, and uses of regular classes and static classes.

1. Regular Class (Non-static Class):

A regular class in Java is the standard way of defining a class. An instance of a regular class must be created using the new keyword. It can have instance variables, constructors, and instance methods.

Key Features of Regular Class:

* Instance Variables: Regular classes can have instance variables (non-static variables), which are associated with individual objects.
* Instance Methods: Regular classes can have methods that belong to individual objects, meaning they operate on instance variables.
* Constructor: A regular class can have constructors to initialize its objects.
* Memory Allocation: Each object created from a regular class gets its own copy of instance variables.

Example of Regular Class:

public class Car {

private String model;

private int year;

// Constructor

public Car(String model, int year) {

this.model = model;

this.year = year;

}

// Instance Method

public void startEngine() {

System.out.println("Starting the engine of " + model + " (" + year + ")");

}

public static void main(String[] args) {

// Creating an instance of the class

Car myCar = new Car("Toyota", 2021);

myCar.startEngine(); // Calling an instance method

}

}

In the example above, the Car class is a regular (non-static) class. To use it, you must create an object (myCar) and then call the methods on that object.

2. Static Class (Inner Static Class in Java):

In Java, static classes are typically inner classes defined with the static keyword. A static class is nested inside another class but behaves differently from non-static inner classes. Static classes can be instantiated without needing an instance of the outer class.

Key Features of Static Class:

* Belongs to the Outer Class: A static class is associated with the outer class, not with an instance of the outer class. It can be accessed and instantiated independently of an object of the outer class.
* No Access to Instance Variables: A static class cannot access instance variables or methods of the outer class because it is not tied to a specific instance of the outer class.
* Memory Efficiency: Static classes do not require an instance of the outer class to be instantiated, making them more memory efficient.
* Used for Utility Classes: Static classes are often used for utility classes or nested classes that don't need an instance of the outer class.

Example of Static Class (Static Inner Class):

public class OuterClass {

// Static class

static class StaticInnerClass {

public void display() {

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

}

}

public static void main(String[] args) {

// Accessing the static inner class without an instance of OuterClass

OuterClass.StaticInnerClass inner = new OuterClass.StaticInnerClass();

inner.display(); // Calling the method of the static inner class

}

}

In the above example, StaticInnerClass is a static nested class within OuterClass. You can create an instance of StaticInnerClass without needing to create an instance of OuterClass.

Differences Between Regular Class and Static Class:

| Feature | Regular Class | Static Class |
| --- | --- | --- |
| Definition | Defined as a normal class. | Defined as a nested class using the static keyword. |
| Instance vs Class | Requires an instance of the class to use it (object-oriented). | Can be accessed without an instance of the outer class (if it's an inner class). |
| Access to Outer Class | Can access instance variables and methods of the outer class (if it's a non-static inner class). | Cannot access instance variables or methods of the outer class. |
| Memory Allocation | Each object has its own set of instance variables. | Only one instance of a static class exists, regardless of the outer class's instances. |
| Instantiation | Must be instantiated using new. | Can be instantiated independently (if it's an inner class, it must be accessed through the outer class). |
| Usage | Used for creating objects with unique states. | Used for utility classes, helpers, or nested classes that do not need access to outer class's instance variables. |
| Example Usage | Regular objects like Car, Person. | Utility methods, static inner classes, nested classes that don't need an instance of the outer class. |

Use Cases:

* Regular Class:
  + When you need to create individual objects with unique states and behavior.
  + Typical use cases include modeling real-world entities (e.g., Car, Person, Book).
* Static Class:
  + When you need to create a class that does not need an instance of the outer class.
  + Used in scenarios where the class provides utility methods or helper methods that don’t require state management.
  + For example, a database utility class inside a class that doesn’t require any instance variables.

Conclusion:

* A regular class represents entities with state and behavior and requires an instance to be created.
* A static class (typically an inner class) does not require an instance of the outer class and is often used for utility purposes or when you need to group related classes without requiring state.

* Memory management

Memory management in Java refers to the process of efficiently allocating and deallocating memory to store and manage objects during the execution of a Java application. Java provides automatic memory management through Garbage Collection (GC), but developers must still follow best practices to optimize memory usage.

Key Aspects of Memory Management in Java:

1. Heap Memory
2. Stack Memory
3. Garbage Collection
4. Memory Leaks
5. Object Lifecycle

1. Heap Memory:

The Heap is the part of memory where objects are stored. It is used for dynamic memory allocation and stores instances of classes, arrays, and other objects.

Key Points:

* The Heap is managed by the Garbage Collector (GC), which automatically frees memory by reclaiming unused objects.
* Memory is allocated on the Heap using new keyword for creating objects.
* The Heap is divided into two parts:
  + Young Generation: Where new objects are created. It includes:
    - Eden Space: Where objects are first allocated.
    - Survivor Space: Objects that survive the initial garbage collection are moved here.
  + Old Generation (Tenured Generation): Where objects that have survived multiple garbage collection cycles are stored.

Example:

class Person {

String name;

int age;

Person(String name, int age) {

this.name = name;

this.age = age;

}

}

public class Main {

public static void main(String[] args) {

Person p = new Person("John", 25); // 'p' is stored in the Heap memory

}

}

In the above example, the Person object is stored in the Heap memory.

2. Stack Memory:

The Stack is used to store local variables, method calls, and references to objects. It is a much smaller memory region compared to the Heap and has a LIFO (Last In, First Out) structure.

Key Points:

* Each time a method is called, a new stack frame is created.
* Local variables and method arguments are stored in the stack, and they are destroyed once the method execution completes.
* The stack is faster than the Heap because of its LIFO structure and because memory is automatically deallocated once the method execution finishes.

Example:

public class StackExample {

public static void main(String[] args) {

int a = 5; // 'a' is stored in Stack memory

int b = 10; // 'b' is stored in Stack memory

}

}

In this example, the local variables a and b are stored in the Stack memory.

3. Garbage Collection (GC):

Garbage Collection is the process by which the JVM (Java Virtual Machine) automatically frees up memory by deleting objects that are no longer in use (i.e., objects that are not referenced by any part of the program).

Key Points about Garbage Collection:

* Automatic Memory Management: The JVM takes care of reclaiming memory occupied by unreachable objects, so developers don’t need to manually deallocate memory.
* Reachability: An object is considered unreachable when there are no references to it from active parts of the program.
* Garbage Collectors: The JVM has several types of garbage collectors, such as:
  + Serial GC: A simple, single-threaded garbage collector suitable for small applications.
  + Parallel GC: Uses multiple threads for GC, improving performance in multi-core systems.
  + CMS (Concurrent Mark-Sweep) GC: Reduces pause times by performing garbage collection concurrently with the application’s execution.
  + G1 (Garbage-First) GC: Aims to provide predictable low pause times by dividing the heap into regions and collecting garbage incrementally.

How Garbage Collection Works:

1. Mark Phase: The garbage collector identifies all live objects.
2. Sweep Phase: It deletes the unreachable objects and frees up memory.
3. Compact Phase: The JVM compacts the memory by moving objects to eliminate fragmentation (in some GC types like G1).

Example:

In the following code, after the line obj = null;, the object p will be eligible for garbage collection since there are no more references to it:

class Person {

String name;

int age;

}

public class Main {

public static void main(String[] args) {

Person p = new Person(); // 'p' is stored in Heap memory

p = null; // The object referenced by 'p' is now eligible for GC

}

}

4. Memory Leaks:

A memory leak occurs when objects are unintentionally retained in memory because they are still referenced, even though they are no longer needed. This can cause a program to consume excessive memory and eventually lead to performance degradation or application crashes.

Causes of Memory Leaks:

* Unclosed resources: For example, not closing database connections, file streams, or sockets.
* Static references: Static fields holding references to objects that should be released.
* Circular references: When two objects reference each other but are no longer needed by the program.

Example of Memory Leak:

class MemoryLeakExample {

private static List<MemoryLeakExample> list = new ArrayList<>();

public void createLeak() {

list.add(new MemoryLeakExample()); // Objects are added to a static list, causing a memory leak.

}

}

Here, objects are being added to a static list that is never cleared, causing them to remain in memory even if they are no longer needed.

5. Object Lifecycle and Memory Management

The lifecycle of an object starts when it is created using the new keyword and ends when it is no longer referenced (and thus eligible for garbage collection).

Object Lifecycle:

1. Creation: Memory for the object is allocated in the heap.
2. Usage: The object is referenced by local or instance variables.
3. Dereferencing: The object is no longer referenced.
4. Garbage Collection: The object is collected by the GC, and its memory is freed.

Memory Management Best Practices:

1. Avoid Memory Leaks:
   * Always release resources like database connections, file streams, and network connections.
   * Nullify references when objects are no longer needed.
2. Use WeakReferences:
   * In some cases, you can use WeakReference to reference an object in such a way that it can be garbage-collected even if the reference still exists.
3. Optimize Object Creation:
   * Avoid creating unnecessary objects in performance-critical code. For instance, reuse objects if possible.
4. Use Proper Garbage Collection Configuration:
   * Select the appropriate garbage collector depending on your application’s requirements. For example, use G1 GC if you need low pause times for large applications.
5. Minimize Static Variables:
   * Static variables can prevent objects from being garbage collected, as they keep objects alive for the lifetime of the application.

Conclusion:

Memory management in Java is largely handled by the JVM, especially with the help of Garbage Collection, but developers must still be mindful of object creation, memory leaks, and efficient resource management. Understanding how memory is allocated in Heap and Stack and how garbage collection works can help in writing more efficient and optimized Java programs

* How to sort Hasmap data based on keys?

To sort a HashMap based on keys in Java, you can convert the HashMap to a TreeMap or use a List of entries and sort them manually using a custom comparator. Here’s how you can do that:

1. Using a TreeMap:

A TreeMap in Java is a SortedMap implementation that automatically sorts its keys. When you create a TreeMap from a HashMap, the data is automatically sorted based on the keys in natural order (if the keys implement Comparable) or according to a custom Comparator.

Example: Using TreeMap to sort by keys

import java.util.\*;

public class SortHashMapByKey {

public static void main(String[] args) {

// Create a HashMap

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

map.put("Banana", 3);

map.put("Apple", 1);

map.put("Orange", 2);

// Convert HashMap to TreeMap (which sorts by keys)

TreeMap<String, Integer> sortedMap = new TreeMap<>(map);

// Print sorted map

System.out.println("Sorted HashMap based on keys: " + sortedMap);

}

}

Output:

Sorted HashMap based on keys: {Apple=1, Banana=3, Orange=2}

In the example above:

* The TreeMap sorts the entries in natural lexicographical order (alphabetical order for String keys).

2. Sorting HashMap entries manually using a List and Comparator:

If you want more control over how the map is sorted (e.g., sorting in reverse order or using a custom comparator), you can:

1. Convert the HashMap to a List of Map.Entry.
2. Sort the List using Collections.sort() with a custom comparator.
3. Optionally, put the sorted entries back into a new map.

Example: Sorting HashMap by keys using List and Comparator

import java.util.\*;

public class SortHashMapByKeyCustom {

public static void main(String[] args) {

// Create a HashMap

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

map.put("Banana", 3);

map.put("Apple", 1);

map.put("Orange", 2);

// Convert the HashMap to a List of Map.Entry

List<Map.Entry<String, Integer>> entryList = new ArrayList<>(map.entrySet());

// Sort the List by key using a custom Comparator

entryList.sort(Map.Entry.comparingByKey()); // Sorting by key in natural order

// Print the sorted map entries

System.out.println("Sorted HashMap based on keys:");

for (Map.Entry<String, Integer> entry : entryList) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

}

}

Output:

Sorted HashMap based on keys:

Apple: 1

Banana: 3

Orange: 2

3. Sorting in Reverse Order:

If you want to sort the HashMap entries in reverse order by key, you can pass a custom comparator to Collections.sort():

import java.util.\*;

public class SortHashMapByKeyReverse {

public static void main(String[] args) {

// Create a HashMap

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

map.put("Banana", 3);

map.put("Apple", 1);

map.put("Orange", 2);

// Convert the HashMap to a List of Map.Entry

List<Map.Entry<String, Integer>> entryList = new ArrayList<>(map.entrySet());

// Sort the List by key in reverse order

entryList.sort(Map.Entry.<String, Integer>comparingByKey().reversed());

// Print the sorted map entries

System.out.println("Sorted HashMap based on keys in reverse order:");

for (Map.Entry<String, Integer> entry : entryList) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

}

}

Output:

Sorted HashMap based on keys in reverse order:

Orange: 2

Banana: 3

Apple: 1

Summary:

* Using TreeMap: This is the simplest way to sort a HashMap by keys. The TreeMap will automatically sort the entries based on their keys (in natural order).
* Using a List and Comparator: This approach allows you to sort the map manually, which provides more flexibility (e.g., sorting in reverse order, or using a custom comparator).

In both cases, you can choose how the keys are sorted (in natural order or using a custom sorting criterion).

* How to sort Hasmap data based on values?

To sort a HashMap based on values in Java, you can follow a similar approach as sorting by keys but with a slight difference in the comparator. Here's how you can do that:

Steps to Sort HashMap Based on Values:

1. Convert the HashMap to a List of entries using the entrySet() method.
2. Sort the List using a custom comparator that compares the values of the entries.
3. Optionally, rebuild the HashMap from the sorted List, but typically you’ll use the List itself for further processing.

Example 1: Sorting HashMap by Values in Ascending Order

import java.util.\*;

public class SortHashMapByValue {

public static void main(String[] args) {

// Create a HashMap

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

map.put("Banana", 3);

map.put("Apple", 1);

map.put("Orange", 2);

// Convert HashMap to List of Map.Entry

List<Map.Entry<String, Integer>> entryList = new ArrayList<>(map.entrySet());

// Sort the list based on values (ascending order)

entryList.sort(Map.Entry.comparingByValue());

// Print the sorted map entries

System.out.println("Sorted HashMap based on values (ascending order):");

for (Map.Entry<String, Integer> entry : entryList) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

}

}

Output:

Sorted HashMap based on values (ascending order):

Apple: 1

Orange: 2

Banana: 3

In this example:

* We used Map.Entry.comparingByValue() to sort the entries based on values in ascending order.

Example 2: Sorting HashMap by Values in Descending Order

If you want to sort the HashMap based on values in descending order, you can modify the comparator to use the reversed() method.java

Copy code

import java.util.\*;

public class SortHashMapByValueDesc {

public static void main(String[] args) {

// Create a HashMap

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

map.put("Banana", 3);

map.put("Apple", 1);

map.put("Orange", 2);

// Convert HashMap to List of Map.Entry

List<Map.Entry<String, Integer>> entryList = new ArrayList<>(map.entrySet());

// Sort the list based on values (descending order)

entryList.sort(Map.Entry.<String, Integer>comparingByValue().reversed());

// Print the sorted map entries

System.out.println("Sorted HashMap based on values (descending order):");

for (Map.Entry<String, Integer> entry : entryList) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

}

}

Output:

Sorted HashMap based on values (descending order):

Banana: 3

Orange: 2

Apple: 1

In this example:

* We used Map.Entry.comparingByValue().reversed() to sort the entries in descending order.

Example 3: Using a Custom Comparator to Sort by Values

If you want to sort by values using a custom sorting logic, you can pass a custom comparator to the sort() method.

import java.util.\*;

public class SortHashMapByValueCustom {

public static void main(String[] args) {

// Create a HashMap

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

map.put("Banana", 3);

map.put("Apple", 1);

map.put("Orange", 2);

// Convert HashMap to List of Map.Entry

List<Map.Entry<String, Integer>> entryList = new ArrayList<>(map.entrySet());

// Sort the list based on a custom comparator (e.g., values in odd/even order)

entryList.sort((entry1, entry2) -> {

// Custom comparator logic (ascending order)

return entry1.getValue() - entry2.getValue();

});

// Print the sorted map entries

System.out.println("Sorted HashMap based on custom comparator:");

for (Map.Entry<String, Integer> entry : entryList) {

System.out.println(entry.getKey() + ": " + entry.getValue());

}

}

}

Output:

Sorted HashMap based on custom comparator:

Apple: 1

Orange: 2

Banana: 3

In this example:

* We used a custom comparator (entry1, entry2) -> entry1.getValue() - entry2.getValue() to sort the map by values in ascending order.

Summary of Sorting by Values:

1. Use Map.Entry.comparingByValue() to sort by values in natural order (ascending).
2. Use reversed() on comparingByValue() to sort in descending order.
3. Custom comparator: You can also define your own sorting logic by providing a custom comparator.

Sorting by values will always involve first converting the HashMap to a List of Map.Entry objects because HashMap does not guarantee any ordering of its elements. Once sorted, you can either keep the data in a List or rebuild the sorted entries back into a LinkedHashMap if you want to retain the order.

Top of Form

Bottom of Form

### Python

* Can oops concept be implemented in python? How to implement hash map in python?
* **Yes**, Python supports OOP concepts (Encapsulation, Inheritance, Polymorphism, and Abstraction) in a very flexible way.
* HashMap **in Python**: Python provides a built-in data structure dict, which is essentially a **hash map** that supports key-value mapping with efficient retrieval. For custom implementation, you can manually create a hash table using lists and handle collisions and resizing as needed.

* Why we prefer Python over Java. What are new features in Python...? \

Summary of Why Python is Preferred Over Java:

1. Ease of Use and Readability: Python has a simpler, more readable syntax, making it easier to write and maintain code.
2. Faster Development: Python's flexibility allows developers to develop applications faster compared to Java.
3. Strong Ecosystem: Python has a rich set of libraries for AI, web development, automation, and more.
4. Cross-Platform: Both Java and Python are cross-platform, but Python's simplicity makes it easier to deploy.
5. Memory Management: Python's automatic memory management reduces the complexity of memory handling.

New Features in Python:

* F-strings for easier string formatting.
* Type Hinting for better code readability and error checking.
* AsyncIO for asynchronous programming.
* Data Classes for reducing boilerplate in class definitions.
* Walrus Operator for assignment in expressions.
* Pattern Matching for more expressive conditional statements.

These features make Python an increasingly powerful and efficient language for modern development across various domains like web development, data science, AI, and more.

Bottom of Form

* Difference between Soap and Rest.

SOAP (Simple Object Access Protocol) and REST (Representational State Transfer) are two different approaches for implementing web services. Here’s a comparison between them based on various parameters:

1. Protocol vs. Architectural Style

* SOAP (Protocol): SOAP is a protocol that defines a set of rules for structuring messages. It is a standardized and more rigid method for exchanging information in the form of XML messages over a network.
* REST (Architectural Style): REST is an architectural style rather than a strict protocol. It provides guidelines for creating web services, which can be used over HTTP using standard operations like GET, POST, PUT, DELETE, etc.

2. Message Format

* SOAP: SOAP messages are always formatted in XML (Extensible Markup Language), which is strict and requires parsing. This adds a layer of complexity but provides greater flexibility.
* REST: REST supports multiple formats, including XML, JSON, Plain Text, HTML, etc. The most common format used is JSON, as it is lightweight and easier to parse.

3. Transport Protocol

* SOAP: SOAP can operate over multiple protocols, including HTTP, SMTP, JMS, and FTP. However, HTTP is the most commonly used transport protocol.
* REST: REST is strictly based on HTTP (HyperText Transfer Protocol) and leverages the existing HTTP methods (GET, POST, PUT, DELETE) for communication between clients and servers.

4. Statefulness

* SOAP: SOAP is generally considered stateful but can be designed to be stateless as well. Each SOAP request can carry its own context, and it is possible to maintain a session across multiple requests.
* REST: REST is stateless, meaning each request from a client to the server must contain all the information needed to understand and process the request (no session is stored on the server side).

5. Complexity

* SOAP: SOAP is relatively more complex because it requires a predefined format and strict rules for communication, which can include complex XML structures, headers, and namespaces.
* REST: REST is simpler and more lightweight. It uses standard HTTP methods and can handle responses in multiple formats (mainly JSON, which is lightweight and easier to work with).

6. Performance

* SOAP: SOAP messages are larger and require more bandwidth and processing power because the messages are always XML-based, which tends to be more verbose and harder to parse.
* REST: REST is faster and more efficient because it typically uses JSON, which is lightweight and easier to parse than XML. This makes REST better for high-performance applications.

7. Security

* SOAP: SOAP has built-in support for security standards like WS-Security, which provides message-level security, including encryption, authentication, and authorization.
* REST: REST does not have built-in security like SOAP. Instead, security is typically handled using HTTPS (SSL/TLS encryption) or other mechanisms like OAuth for authentication and authorization.

8. Error Handling

* SOAP: SOAP has a well-defined error handling mechanism. If something goes wrong, SOAP returns a standardized fault message with an error code and description.
* REST: REST uses standard HTTP status codes (like 200 for success, 404 for not found, 500 for server error) for error handling. While not as standardized as SOAP’s error handling, it’s easy to implement and understand.

9. Use Cases

* SOAP: SOAP is better suited for enterprise-level applications, complex transactions, and scenarios requiring high security (e.g., banking systems, payment gateways). It is ideal for applications that need strict messaging protocols and advanced security features.
* REST: REST is ideal for web-based applications, mobile apps, and lightweight services where speed, scalability, and flexibility are key. It is widely used in modern web development, APIs, and microservices.

10. Caching

* SOAP: SOAP doesn’t support caching as it is a complex, stateful protocol.
* REST: REST is designed to be stateless and can take full advantage of HTTP caching mechanisms, which can improve performance by reducing server load and response time.

11. Standardization

* SOAP: SOAP is highly standardized. It follows specific standards like WS-Security, WS-Addressing, WS-ReliableMessaging, and WS-Coordination that provide comprehensive capabilities for web services.
* REST: REST is not bound by a strict set of standards. It follows guidelines and constraints (like statelessness, cacheability, etc.) but doesn’t require the use of a specific message format or security standard.

12. Tool Support

* SOAP: SOAP has strong tool support for development, especially in enterprise environments. Many development frameworks and platforms offer support for generating SOAP clients and servers.
* REST: REST is supported by almost all modern web development frameworks, and it is easier to work with using tools like Postman, Swagger, etc., due to its simplicity and use of HTTP methods.

Summary of Differences:

| Feature | SOAP | REST |
| --- | --- | --- |
| Protocol vs Style | Protocol | Architectural style |
| Message Format | XML | JSON, XML, HTML, Plain Text |
| Transport | HTTP, SMTP, JMS, FTP, etc. | HTTP |
| Statefulness | Can be stateful or stateless | Stateless |
| Complexity | More complex | Simpler |
| Performance | Slower due to XML processing | Faster (JSON) |
| Security | Built-in (WS-Security) | Handled via HTTPS, OAuth, etc. |
| Error Handling | Fault codes in SOAP envelope | HTTP status codes |
| Caching | Limited caching support | Built-in HTTP caching support |
| Use Cases | Enterprise applications, secure systems | Web apps, mobile apps, public APIs |

When to Use SOAP vs REST:

* Use SOAP when:
  + You need advanced security and transaction management.
  + You need reliable messaging and strict standards (e.g., banking, payment systems).
  + Your application requires stateful operations or complex interactions.
* Use REST when:
  + You need a lightweight, fast solution.
  + You are building web applications, mobile apps, or public APIs.
  + You need scalability, flexibility, and easy integration with various systems.

* Difference between put and post

In the context of HTTP methods, PUT and POST are both used to send data to a server, but they are used for different purposes and behave differently. Here's a detailed comparison between PUT and POST:

1. Purpose:

* PUT:
  + Idempotent operation. It is used to update an existing resource or create a resource at a specific URL if it doesn’t exist.
  + If you send the same PUT request multiple times with the same data, the result will always be the same (i.e., the resource will be updated to the same state).
  + Typically used to update a resource or create a resource at a known URI.
* POST:
  + Non-idempotent operation. It is used to submit data to be processed, such as creating a new resource, or performing an action.
  + Sending the same POST request multiple times might result in different outcomes (e.g., creating multiple resources or triggering actions repeatedly).
  + Typically used for creating new resources, or triggering processes, such as submitting forms.

2. Idempotency:

* PUT:
  + Idempotent: Repeating the same PUT request results in the same state of the resource, so it can be safely repeated without side effects.
* POST:
  + Non-idempotent: Repeating a POST request may lead to different results each time (e.g., creating multiple resources).

3. Resource Creation:

* PUT:
  + If you send a PUT request to a specific URI, it will either create the resource (if it doesn't already exist) or update the resource (if it already exists).
  + The client typically specifies the URI of the resource.
* POST:
  + POST is used when the server is responsible for determining the URI of the newly created resource.
  + It is used to create new resources, but the server generates the resource URI.

4. Data Handling:

* PUT:
  + PUT typically sends the entire resource (or a full update) in the request body.
  + The data sent with a PUT request completely replaces the current resource.
* POST:
  + POST is used to send data to the server for processing, such as form submission, or batch processing.
  + It can handle partial updates or new resource creation, and it doesn't necessarily replace the existing resource.

5. Example Use Cases:

* PUT:
  + Updating a user’s details (e.g., changing a user's address).
  + Replacing the contents of a file or document on the server.
  + Example: PUT /users/123 might update the user with ID 123.
* POST:
  + Creating a new user or submitting a form.
  + Uploading a file to the server.
  + Example: POST /users might create a new user.

6. HTTP Status Codes:

* PUT:
  + 200 OK: The resource was successfully updated.
  + 201 Created: If the resource is being created for the first time at a given URL.
  + 204 No Content: If the update is successful but there is no content to return.
* POST:
  + 200 OK: The resource was successfully created or processed.
  + 201 Created: The resource was successfully created (usually with a Location header indicating the URI of the new resource).
  + 400 Bad Request: The request was malformed.
  + 404 Not Found: The requested resource cannot be found.

7. Caching:

* PUT:
  + PUT is typically not cached because it generally represents an update to an existing resource.
* POST:
  + POST is also not cached by default, but it may be cacheable depending on the implementation and usage.

8. Security and Control:

* PUT:
  + PUT is usually used when a client wants to directly control the resource's state at a specific URL.
* POST:
  + POST is often used for actions or when a server controls the resource creation, and the client doesn't necessarily know the final resource URI.

Summary of Differences:

| Feature | PUT | POST |
| --- | --- | --- |
| Purpose | Update or create a resource at a URI | Submit data to be processed, create resources |
| Idempotency | Idempotent (repeated calls have same result) | Non-idempotent (repeated calls can have different results) |
| Data Handling | Replaces the entire resource | Sends data for creation or processing |
| Resource URI | Client specifies URI | Server generates the URI |
| Use Case | Updating resources, replacing resources | Creating new resources, form submission, batch processing |
| Common Status Codes | 200 OK, 201 Created, 204 No Content | 200 OK, 201 Created, 400 Bad Request |
| Cacheability | Not typically cached | Not typically cached |

When to Use PUT vs. POST:

* Use PUT when:
  + You want to update or replace a resource at a specific URI.
  + You want the operation to be idempotent (repeating the same request will result in the same outcome).
* Use POST when:
  + You are creating a new resource or submitting data to be processed.
  + You want the operation to be non-idempotent (repeating the same request could create multiple resources or trigger repeated actions).

* What is api? what is rest api?

What is an API?

API stands for Application Programming Interface. It is a set of rules, protocols, and tools that allows different software applications to communicate with each other. An API defines the methods and data formats that applications can use to request and exchange information. APIs enable developers to integrate and extend functionalities from different software systems, services, or platforms without needing to understand their internal workings.

In simpler terms, an API acts as an intermediary between different software components, allowing them to interact with each other.

Types of APIs:

* Web APIs: These allow applications to communicate over the internet.
* Library APIs: These provide a set of functions or routines for software libraries to interact with other software.
* Operating System APIs: These allow applications to interact with the operating system.

What is a REST API?

A REST API (Representational State Transfer Application Programming Interface) is a type of web API that follows the principles and constraints of REST (an architectural style) to provide services over the internet. REST APIs are widely used due to their simplicity, scalability, and ease of use with web-based applications.

Key Characteristics of a REST API:

1. Stateless:
   * Each request from a client to a server must contain all the information needed to understand and process the request (such as parameters, authentication, etc.).
   * The server does not store any state between requests. Every request is treated independently.
2. Client-Server Architecture:
   * A RESTful system operates on a client-server model, where the client and server are separate entities that communicate over a network (usually HTTP).
   * The client is responsible for the user interface and user experience, while the server is responsible for processing requests and managing data.
3. Uniform Interface:
   * REST APIs use a standardized, uniform set of operations. The most common HTTP methods used in REST APIs are:
     + GET: Retrieve information (data) from the server.
     + POST: Create new resources on the server.
     + PUT: Update existing resources on the server.
     + DELETE: Delete resources from the server.
4. Stateless Communication:
   * Each API call is independent, meaning the server doesn't retain any information about the client's state between calls.
5. Representation of Resources:
   * In REST, resources (data objects or services) are represented as URLs. Clients interact with these resources by using HTTP methods.
   * Resources can be represented in different formats such as JSON, XML, HTML, etc.
6. Cacheable:
   * Responses from the server can be explicitly marked as cacheable or non-cacheable, allowing clients to store responses and reduce the need for repeated requests.
7. Layered System:
   * A REST API can be organized in layers, where each layer performs a specific function, such as security, load balancing, etc. The client doesn't need to know whether it's interacting directly with the server or a proxy.

How REST API Works:

1. Client sends an HTTP request to the server with the required data, method, and headers.
2. The server processes the request and retrieves or manipulates data based on the client’s requirements.
3. The server returns an HTTP response, usually with data in JSON or XML format, containing the results of the request.

REST API vs Other Web Services (SOAP, etc.):

* Simplicity: REST is simpler and uses standard HTTP methods (GET, POST, PUT, DELETE). SOAP, on the other hand, is a more rigid, XML-based protocol.
* Data Format: REST supports multiple data formats such as JSON, XML, or even plain text, with JSON being the most common. SOAP, on the other hand, exclusively uses XML.
* Performance: REST is generally considered faster and more lightweight because it uses simpler message formats (like JSON), while SOAP can be heavier due to the XML-based messages.
* Statefulness: REST is stateless, while SOAP can be stateful.

Benefits of REST APIs:

* Scalability: The stateless nature of REST allows it to scale more efficiently.
* Ease of Use: REST APIs are simple to implement and use HTTP as the underlying protocol, which is widely supported by various systems.
* Flexibility: REST APIs can work with a variety of formats and clients, including web browsers, mobile devices, and desktop applications.
* Performance: Because REST is lightweight (especially with JSON), it generally performs faster than SOAP-based web services.

REST API Example:

For instance, if we wanted to fetch information about a user with an ID of 123 from a server using a REST API, we might use an HTTP GET request like this:

arduino

Copy code

GET https://api.example.com/users/123

If we wanted to create a new user, we might use a POST request:

bash

Copy code

POST https://api.example.com/users

Content-Type: application/json

Body: {"name": "John", "email": "john@example.com"}

Summary:

| Feature | API | REST API |
| --- | --- | --- |
| Definition | Set of rules for software communication | A type of web API following REST principles |
| Protocol | Various (HTTP, SMTP, etc.) | HTTP-based |
| State | Can be stateful or stateless | Stateless |
| Communication | Broad (can use different protocols) | Uses standard HTTP methods (GET, POST, PUT, DELETE) |
| Data Format | Varies (XML, JSON, etc.) | JSON, XML, HTML, etc. |
| Usage | General-purpose communication | Mainly for web-based applications |

In essence, API is a broader concept, while REST API is a specific implementation of API based on REST principles, often used for web services and modern web applications.

* What are method is Rest API like post, put and etc.

In REST API, HTTP methods are used to perform various operations on resources (data objects or services). These methods correspond to CRUD (Create, Read, Update, Delete) operations. Below are the most commonly used HTTP methods in REST APIs:

1. GET

* Purpose: Retrieve data from the server.
* Action: It fetches the resource(s) from the server without making any changes to them.
* Idempotent: Yes. Repeating the same GET request will always return the same data (unless the data changes on the server).

Example:

GET /users/123

This request retrieves the details of the user with ID 123 from the server.

2. POST

* Purpose: Submit data to the server, often to create a new resource.
* Action: Used to create a new resource or submit data for processing. The server generates a new resource (like a new record) based on the data submitted.
* Idempotent: No. Repeating the same POST request will usually create a new resource or trigger an action.

Example:

http

Copy code

POST /users

Content-Type: application/json

Body: {"name": "John", "email": "john@example.com"}

This request creates a new user resource on the server with the data provided in the body of the request.

3. PUT

* Purpose: Update an existing resource or create a new resource at a specific URI.
* Action: Replaces the current resource with the data provided in the request body. If the resource does not exist, it can create a new resource.
* Idempotent: Yes. Repeating the same PUT request with the same data will result in the same resource state.

Example:

http

Copy code

PUT /users/123

Content-Type: application/json

Body: {"name": "John Doe", "email": "john.doe@example.com"}

This request updates the user with ID 123 to have the new name and email.

4. DELETE

* Purpose: Remove a resource from the server.
* Action: Deletes the specified resource.
* Idempotent: Yes. Repeating the same DELETE request on the same resource will have no further effect after the resource is deleted.

Example:

http

Copy code

DELETE /users/123

This request deletes the user with ID 123 from the server.

5. PATCH

* Purpose: Partially update a resource.
* Action: Similar to PUT but used for partial updates rather than replacing the entire resource. It modifies only the fields provided in the request body.
* Idempotent: Yes. Repeating the same PATCH request will have the same result as long as the data doesn’t change.

Example:

http

Copy code

PATCH /users/123

Content-Type: application/json

Body: {"email": "new.email@example.com"}

This request updates only the email address of the user with ID 123.

6. OPTIONS

* Purpose: Retrieve supported HTTP methods for a specific resource or endpoint.
* Action: This method is used to find out which HTTP methods are supported by the server for a particular resource.
* Idempotent: Yes.

Example:

http

Copy code

OPTIONS /users/123

This request returns the HTTP methods allowed for the /users/123 resource (such as GET, PUT, DELETE, etc.).

7. HEAD

* Purpose: Similar to GET, but only retrieves the headers, not the body of the resource.
* Action: This method is used when you want to know about the resource (such as its metadata) without downloading the actual data.
* Idempotent: Yes.

Example:

http

Copy code

HEAD /users/123

This request retrieves the headers (like content-type, last-modified) for the user with ID 123 without fetching the actual content.

Summary of HTTP Methods in REST API:

| HTTP Method | Purpose | Action | Idempotent? |
| --- | --- | --- | --- |
| GET | Retrieve resource(s) | Fetch data from the server without modifying it. | Yes |
| POST | Create a new resource or submit data for processing | Create a new resource or trigger an action. | No |
| PUT | Update or create a resource at a specific URI | Replace a resource or create a resource at a specific location. | Yes |
| DELETE | Remove a resource | Delete a resource from the server. | Yes |
| PATCH | Partially update a resource | Modify only the provided fields of a resource. | Yes |
| OPTIONS | Retrieve supported HTTP methods for a resource | Get the allowed HTTP methods for a given resource. | Yes |
| HEAD | Retrieve headers (no body) | Get the metadata or headers of a resource. | Yes |

These methods allow clients to perform CRUD operations on resources in a RESTful manner, ensuring that web applications can communicate effectively over the web using standard HTTP protocols.

Q). What is patches?

PATCH is an HTTP method used in REST APIs for partially updating a resource. Unlike the PUT method, which replaces the entire resource, PATCH modifies only the specific fields of a resource that are provided in the request.

Key Points:

* Purpose: Update part of a resource (only specified fields).
* Action: Modify the resource without replacing it entirely.
* Idempotent: Yes (repeated requests with the same data have the same result).
* Use Case: When only a subset of fields needs to be updated, e.g., changing just one field in a user's profile instead of updating the entire profile.

Example:

Suppose you have a resource representing a user:

json

Copy code

{

"id": 123,

"name": "John",

"email": "john@example.com"

}

To update only the email using PATCH:

http

Copy code

PATCH /users/123

Content-Type: application/json

Body: {"email": "new.email@example.com"}

This would update just the email field of the user with ID 123, without altering other fields (like the name).

* Difference between Json and XML

1. Format

* JSON (JavaScript Object Notation): Lightweight, text-based, easy to read and write. Represents data as key-value pairs.
* XML (Extensible Markup Language): A markup language that uses tags to define elements and attributes. It's more verbose and complex.

2. Syntax

* JSON:
  + Uses key-value pairs enclosed in curly braces ({}).
  + Arrays are enclosed in square brackets ([]).
  + Data is simple and minimal.

Example:

{

"name": "John",

"age": 30

}

* XML:
  + Uses opening and closing tags (<tag>value</tag>).
  + Can include attributes in tags.
  + More verbose due to closing tags.

Example:

<person>

<name>John</name>

<age>30</age>

</person>

3. Data Structure

* JSON:
  + Simple data structures (objects, arrays, numbers, strings).
  + No concept of attributes.
* XML:
  + Hierarchical structure with elements and attributes.
  + Supports mixed content (elements containing both text and child elements).

4. Readability

* JSON: Easier for humans to read and write due to its simpler structure.
* XML: Can be harder to read due to its verbose nature and tag-based structure.

5. Size

* JSON: More compact as it does not require closing tags and uses fewer characters.
* XML: Larger due to the necessity of closing tags and verbose nature.

6. Data Types

* JSON: Supports simple data types like numbers, booleans, and strings.
* XML: All data is represented as text. You may need additional attributes or custom handling for complex types.

7. Parsing and Processing

* JSON: Easier and faster to parse, especially for JavaScript-based applications.
* XML: Requires more processing and can be slower to parse, especially with large documents.

8. Use Cases

* JSON: Preferred in web services and APIs (especially with JavaScript, Node.js, etc.), modern web development, and mobile apps.
* XML: Often used in legacy systems, document-based structures, or where rich data representation is needed (e.g., SOAP-based APIs).

Summary Table:

| Feature | JSON | XML |
| --- | --- | --- |
| Format | Key-value pairs | Tags and attributes |
| Syntax | Simple, minimal | Verbose, with opening/closing tags |
| Readability | Easier to read and write | Harder to read due to verbosity |
| Size | Compact | Larger due to tags |
| Data Structure | Objects, arrays, strings, numbers | Hierarchical, tags with attributes |
| Parsing | Faster, simpler | Slower, more complex parsing |
| Use Cases | Web APIs, JavaScript, mobile | Document storage, legacy systems |

Conclusion:

* JSON is preferred for modern applications due to its simplicity, smaller size, and ease of parsing.
* XML is useful in situations where detailed, structured data is required or when working with legacy systems.

Extra ques added

* count the number of Islands.
* Complexity and optimization
* Difference between sql and nosql
* Group by, order by in sql
* Joins
* Hashmap vs hashset
* Stack vs queue,
* Have you used stack or queue somewhere in the code, directly or indirectly
* Why java is object oriented
* Interface vs class
* Can we write methods in interface, why do we use that when we already have class

* Given an array
* A[] = {1,2,3,4,5}

* Find the min and Max pairs
* How to find the smallest number in a list and what sorting algo. to use.
* difference between C and C++
* what is oops
* explain 4 pillar of oops
* what is difference between jaba and c
* different types of inheritance in C++
* Whag are access specifiers
* what is virtual function
* what is friend function
* what is class and object
* error hierarchy in C++
* different types of polymorphism
* asked a question of system design that I didn't know
* difference between new and malloc
* difference between set and list
* difference between list and array

# Coding & Algorithm:-

* Find if the cycle is present in LinkedList.

* Linked list operations & how u can add last element to the linked list.

* You have given 5GB of files to sort them but you can use only 2 GB at a time. What sorting algorithm you will be using?

* Implement a queue using arrays.

* Convert string to camelcase words separated by space

* Code:-Search an element

* Sort array based on the frequency
* Merging 2 lists and returning sorted list
* Finding kth smallest element after insertion of new element in the list.

* Sort linkedlist using keys

* Sort linkedlist using values

* How node works in linkedlist (reference of tail)? Implement Stack using Queue and Queue using Stack.

* Find the number using binary search.

* Print your name with its character index

* Program to reverse a string.

* Write program for linked list operation using recursion

* In a given string, return the character that appears the minimum number of times in the string. The string will only contain ascii characters

* Difference between list and linked list.

* Merge sort implementation.

* What-is-backtreaking?

* 3 sum leetcode

* Print all permutation of string

* Find Maximum sum and minimum sum in an Array

* Stack vs queue?

* What is the difference between Dynamic Programming and Greedy algorithm?

* When we use DP and when greedy approach?

* Given 2\*2 matrix find the difference of sum of the diagonals Optimize the approach

* print Maximum sum subarray. Pre order, postorder, inorder

* Diagonal difference in Matrix

* Char count in string

* Print next palindrome

* Binary tree, Binary search tree, priority queue

* Balanced binary search tree and diff b/w them