**Section: String**

**Que: What is string in Java?**

**Ans:** String is one of the most common objects used in java programming. These are the sequence of characters. String is the object of ‘String ’class (built-in class) in ‘java.lang’ package. But it is also used as data type.

**Que: What are string Literals?**

**Ans:** String literals are sequences of characters enclosed within double quotes in Java source code. They represent constant values of type String.

String str1 = "Hello, World!";

String str2 = "Java is awesome!";

**Que: Why Strings are Immutable in Java?**

**Ans:** Strings are immutable in Java, meaning once a string object is created, its value cannot be changed. However, we can only modify the reference to the string object. You can create new strings by performing various operations on existing strings.

Strings are immutable in Java for several reasons, including:

**1. Thread Safety:** Immutable objects are inherently thread-safe because their state cannot be modified once they are created. This makes it easier to reason about concurrent code and eliminates the need for synchronization.

**2. Caching**: Java can cache string literals because they are immutable. This means that if you have multiple references to the same string literal, they can all refer to the same memory location, saving memory.

**3. Security:** Strings are often used in hashing and cryptographic operations. Immutability ensures that once a string is created, its value cannot be changed, which helps maintain data integrity in security-sensitive operations.

**4. Performance:** Immutable objects can be optimized by the Java Virtual Machine (JVM). For example, Java can use string interning to avoid unnecessary duplication of string objects, which can improve performance and reduce memory usage.

**5. Design Simplicity**: Immutability simplifies the design of classes and APIs. When you know that a string won't change, you don't need to worry about unexpected modifications to its state, making code easier to understand and maintain.

By making strings immutable, Java promotes good programming practices and helps developers write more reliable and robust code.

**Que: How to declare a string in Java?**

**Ans:** In Java, there are two main ways to declare a string:

**1. Using String Literals:** This is the most common and straightforward way to declare a string. You enclose the sequence of characters within double quotes.

**String str1 = "Hello, World!";**

**2. Using the String Class Constructor**: You can also create a string by instantiating an object of the String class using its constructor. This constructor takes a string as an argument. For example:

**String str2 = new String("Java is awesome!");**

When you create a string using the new keyword along with the String class constructor, it creates a new instance of the String class, just like creating an object of any other class. However, it's important to note that in Java, strings created using the new keyword with the String class constructor are stored in the heap memory, unlike string literals which are stored in the string pool (a special area of the heap memory).

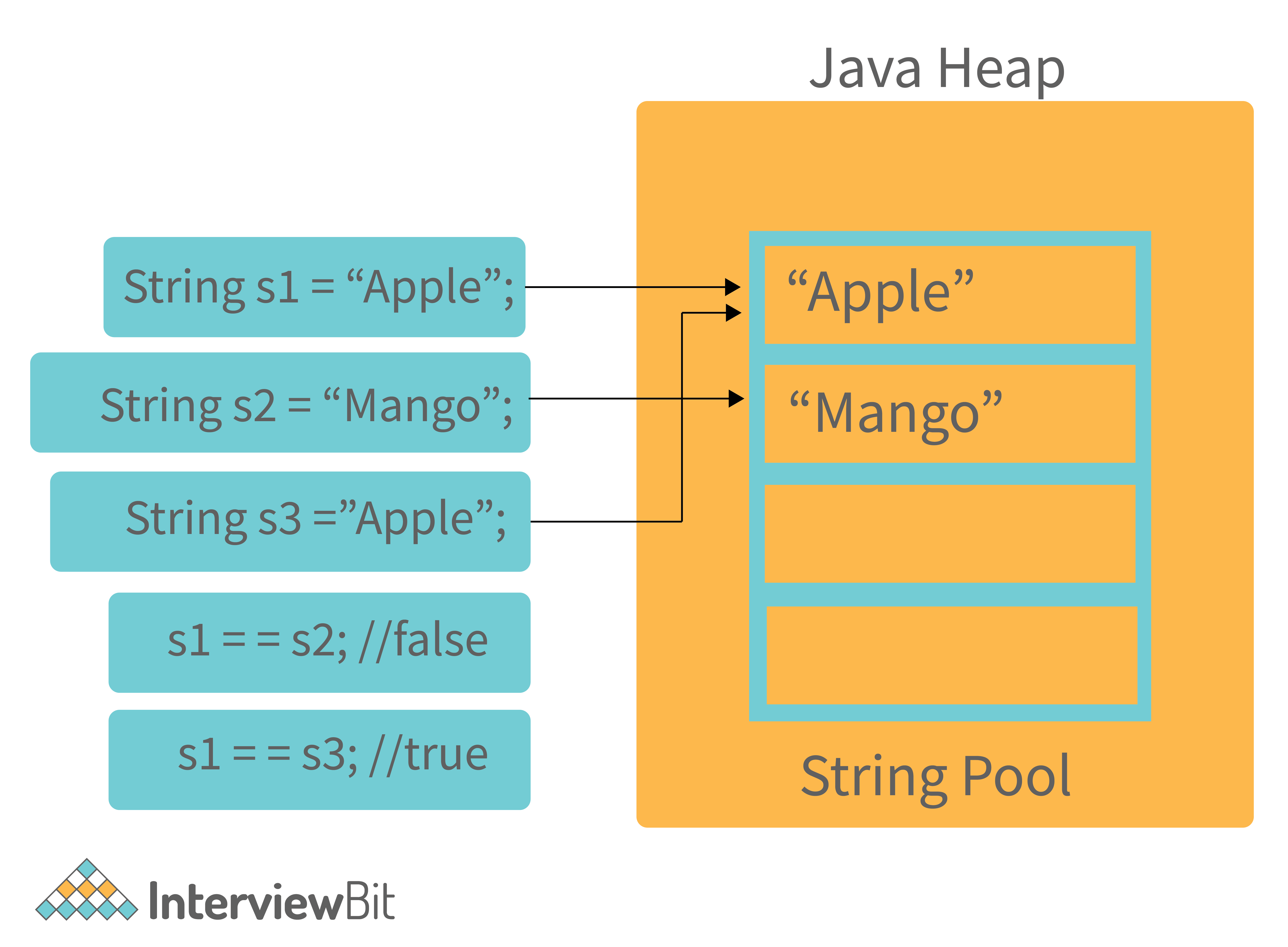
**Que: What is String interning?**

**Ans:** String literals are treated specially by the Java compiler. When the compiler encounters a string literal, it creates a String object with the specified value in the string pool, which is a special area of the heap memory. If another string literal with the same value is encountered later in the code, the compiler will reuse the same String object from the string pool, rather than creating a new one. This process is known as string interning.

It's worth mentioning that using the new keyword with the String constructor is less common than creating strings using string literals ("..."). String literals are more efficient because they are automatically interned.

**Que: Explain String pool in Java.**

**Ans:** String Pool, also known as SCP (String Constant Pool), is a special storage space in Java heap memory that is used to store unique string objects. Whenever a string object is created, it first checks whether the String object with the same string value is already present in the String pool or not, and if it is available, then the reference to the string object from the string pool is returned. Otherwise, the new string object is added to the string pool, and the respective reference will be returned.

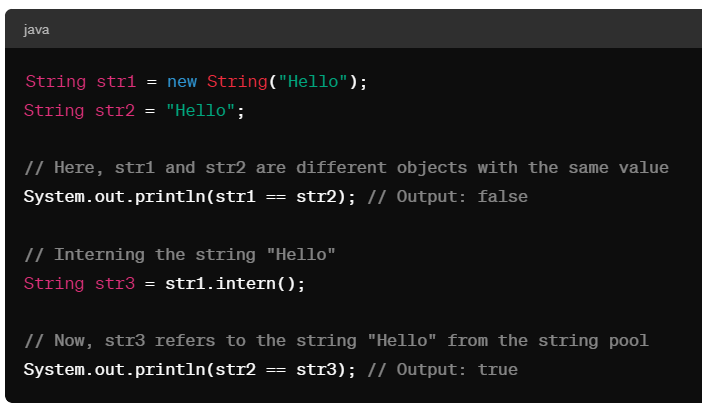


**Que: What does the string intern() method do in Java?**

**Ans:** In Java, the intern() method is a method of the String class. It's used to place the string in the string pool if it's not already present.

Here's how it works:

* If a string equal to this string is already in the pool, then the string from the pool is returned. Otherwise, this string object is added to the pool, and a reference to this string object is returned.
* It's important to note that calling intern() on a string returns a reference to a string from the pool. If the string is already in the pool, the reference returned by intern() will be the same as the reference of the original string. If the string is not in the pool, it will be added, and a reference to the newly added string will be returned.

****

In this example, str1 is created using the new keyword, so it's stored in the heap memory, while str2 is a string literal, so it's stored in the string pool. By calling intern() on str1, we make it refer to the same string object in the string pool as str2. As a result, str2 == str3 returns true.

**Que:** **Is String a primitive or derived type in Java?**

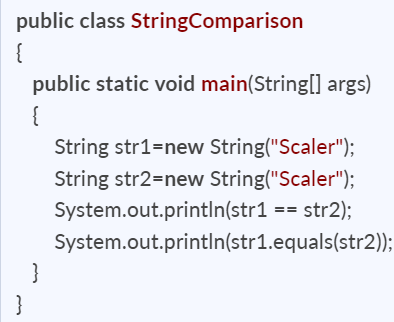
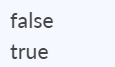
**Ans:** Strings are derived data types. Strings are Java objects that represent sequences of characters. String objects are created using the java.lang.String class. There are many functions that need to be called upon when processing a string, such as substring(), indexof(), equals(), toUppercase(), etc, which primitives types do not have.

**Que: What is the difference between str1 == str2 and str1.equals(str2)?**

**Ans:** Java offers both the equals() method and the "==" operator for comparing objects. However, here are some differences between the two:

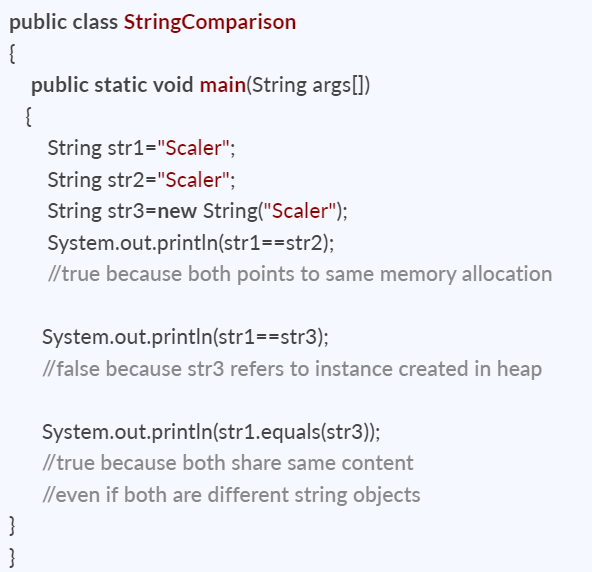
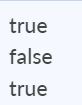
Essentially, equals() is a method, while == is an operator.

The == operator can be used for comparing references (addresses) and the .equals() method can be used to compare content. To put it simply, == checks if the objects point to the same memory location, whereas .equals() compares the values of the objects.

🡪 Output 🡪

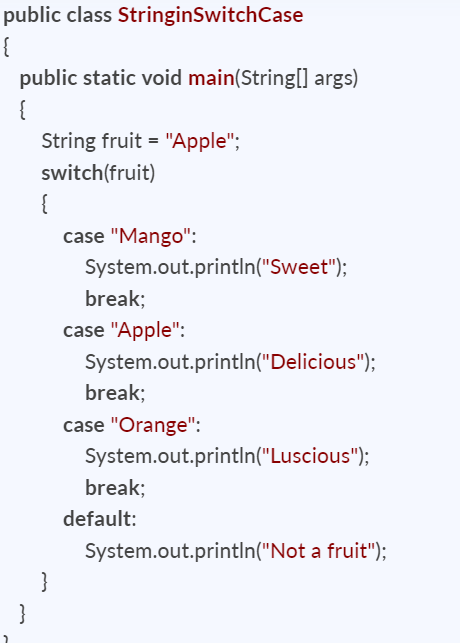
**Que: It possible to compare Strings using the == operator? If so, what is the risk involved?**

**Ans:** Yes, you can compare strings using the == operator. One can use == operators for reference comparison (address comparison). The majority of the time, developers compare strings with the == operator, instead of using the equals() method, resulting in an error.

**🡪** ****

**Que: Can we use a string in the switch case in java?**

**Ans:** Yes, Java allows you to use strings in switch case conditions. Below is a Java program that shows the use of string in switch case.

🡪 Delicious

**Que: Is String thread-safe in Java?**

**Ans:** Strings are immutable objects, which means they can't be changed or altered once they've been created. As a result, whenever we manipulate a String object, it creates a new String rather than modifying the original string object. In Java, every immutable object is thread-safe, which means String is also thread-safe. As a result, multiple threads can access a string. For instance, if a thread modifies the value of a string, instead of modifying the existing one, a new String is created, and therefore, the original string object that was shared among the threads remains unchanged.

**Que: Why is a string used as a HashMap key in Java?**

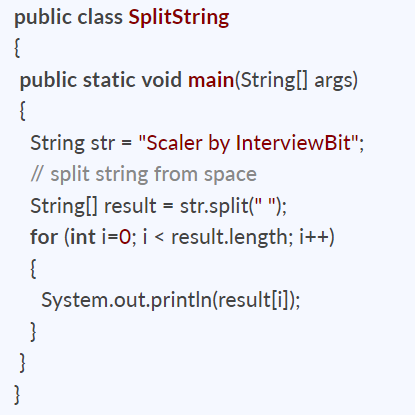
**Ans:** Basically, the HashMap object can store key-value pairs. When creating a HashMap object and storing a key-value pair in that object, you will notice that while storing, the hash code of the key will be calculated, and its calculated value will be placed as the resultant hash code of the key. Now, when the key is passed to fetch its value, then the hash code of the key is calculated again, and if it's equal to the value of the hash code initially calculated, the initial value placed as the resultant hash code of the key is retrieved or fetched.

Let's say we utilized a variable as a key to store data and then changed the value of that variable. In this case, since we have altered the key, the hash code calculated of the current key will not match the hash code at which its value was originally stored. This makes retrieval impossible. String values are immutable, so once they've been created, they can't be changed. As a result, it is recommended to use Strings as HashMap keys.

**Que: What is the best way to split a string in Java?**

**Ans:** Split() is a Java method for breaking a string based on a Java string delimiter (specified regex). For example, a space or a comma(,) will usually be used as the Java string split attribute to break or split the string.

Syntax: string.split(String regex, int limit)



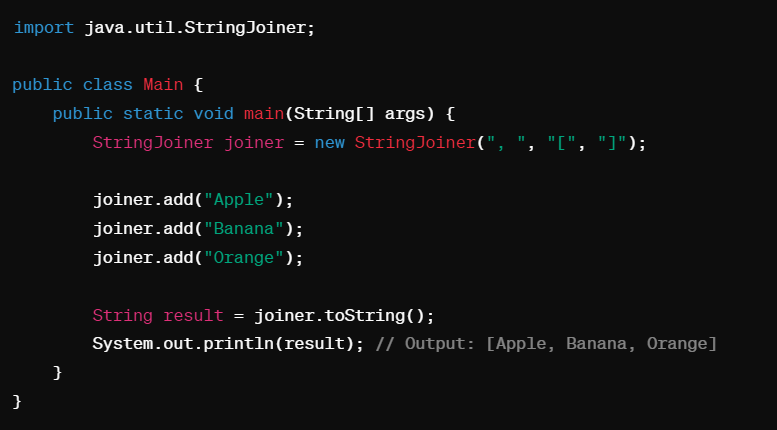
**Que: Why char array is preferred over a String in storing passwords?**

**Ans:** There are various reasons why a char array rather than a string should be used to store passwords. The following are a few of them:

* Strings are immutable: The content of Strings cannot be modified/overwritten because any modification will result in the creation of a new String. As a result, we should always save sensitive data like passwords, Social Security numbers, and so on in a char[] array rather than a String.
* Security: Because String is immutable, storing the password as plain text keeps it in memory until it is cleaned up by the garbage collector. As string uses SCP (String Constant Pool) for re-usability of a string, it's possible that it'll remain in memory for a long time, and anyone with access to the SCP or memory dump can simply identify or retrieve the password in plain text. That's another reason why we should use an encrypted password instead of plain text.
* Logfile safety: With an array, the data can be erased or wiped up, overwritten and the password will not be present anywhere in the system. Whereas, when using plain String, the chances of mistakenly printing the password to monitors, logs, or other insecure locations are substantially higher.

**Que: What do you meant by StringJoiner ?**

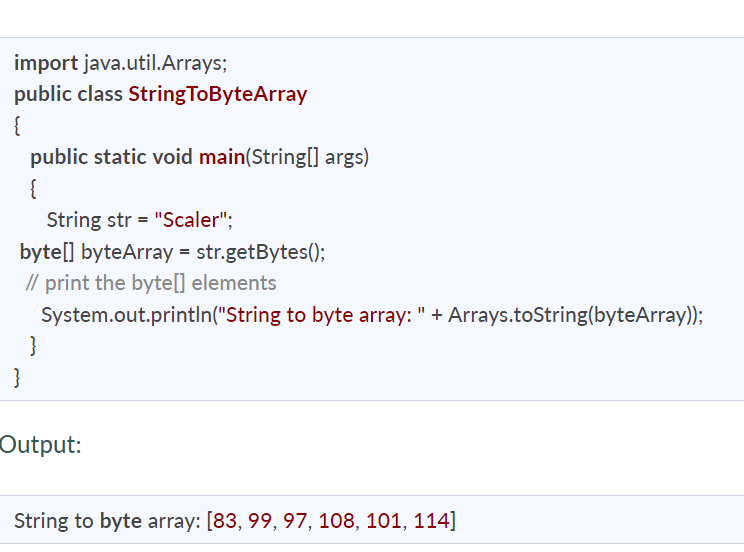
**Ans:** StringJoiner is a class introduced in Java 8 under the java.util package. It provides a convenient way to join strings with a delimiter and optionally a prefix and a suffix. This class is useful when you need to concatenate multiple strings with a specific separator.



* We create a StringJoiner object named joiner with a delimiter ", ", a prefix "[", and a suffix "]".
* We then add strings "Apple", "Banana", and "Orange" to the StringJoiner.
* Finally, we call toString() to get the concatenated string with the specified delimiter, prefix, and suffix.
* StringJoiner provides several constructors to specify different combinations of delimiter, prefix, and suffix. Additionally, you can also set an empty value to be used when there are no elements to join.
* StringJoiner is particularly useful in scenarios where you need to dynamically construct a string by concatenating multiple substrings with a separator, such as building CSV strings, SQL query parts, or constructing messages. It provides a cleaner and more efficient alternative to concatenating strings manually or using StringBuilder when dealing with multiple string parts.

**Que: How can a Java string be converted into a byte array?**

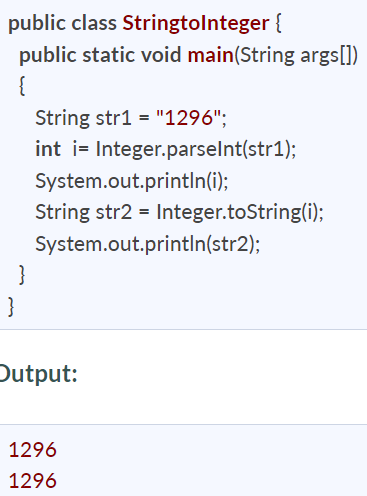
**Ans:** The getBytes() method allows you to convert a string to a byte array by encoding or converting the specified string into a sequence of bytes using the default charset of the platform. Below is a Java program to convert a Java String to a byte array.



**Que: In Java, how do you convert a string to an integer and vice versa?**

**Ans:** There is an Integer class in the Java lang package that provides different methods for converting strings to integers and vice versa. The parseInt() and ValueOf() method allows you to convert a String into an integer and the toString() method allows you to convert an Integer into a String.

Both **parseInt()** and **valueOf()** methods will throw a **NumberFormatException** if the string cannot be parsed as an integer.



**Que: Difference between String, String Builder and String buffer ?**

**Ans:** The main differences between String, StringBuilder, and StringBuffer in Java are related to their immutability, performance, and thread safety.

**1. Immutability:**

* String: Immutable. Once a String object is created, its value cannot be changed. Any operation that seems to modify a String actually creates a new String object.
* StringBuilder: Mutable. StringBuilder allows for the modification of its contents without creating a new object. It is suitable for scenarios where frequent string manipulations are required.
* StringBuffer: Mutable. Similar to StringBuilder, StringBuffer allows for mutable string operations. However, StringBuffer is synchronized, making it thread-safe but potentially slower in performance compared to StringBuilder.

**2. Thread Safety:**

* String: Immutable and inherently thread-safe. Multiple threads can safely read the same String object without causing any issues.
* StringBuilder: Not thread-safe. It means that StringBuilder methods are not synchronized, so they are faster but not suitable for concurrent (multi-threaded) environments.
* StringBuffer: Thread-safe. StringBuffer methods are synchronized, making them safer for use in multi-threaded applications where multiple threads may access or modify the same StringBuffer object concurrently.

**3. Performance:**

* String: Immutable strings are efficient for operations that involve reading or referencing existing strings. However, string concatenation or modifications can result in the creation of multiple intermediate string objects, leading to potential performance overhead.
* StringBuilder: Efficient for string manipulations due to its mutable nature. StringBuilder is preferred for tasks involving frequent string modifications, such as concatenation, appending, or replacing characters in a string.
* StringBuffer: Similar to StringBuilder in functionality but slightly slower due to synchronization. Use StringBuffer in multi-threaded environments where thread safety is required.

**4. Usage**:

* Use String for immutable strings or scenarios where string values remain constant.
* Use StringBuilder for mutable string operations in single-threaded environments or when thread safety is not a concern.
* Use StringBuffer for mutable string operations in multi-threaded environments where thread safety is required.

**Que: How can we convert string to StringBuilder?**

**Ans**: In Java, you can convert a String to a StringBuilder using the StringBuilder class's constructor or the append() method. Here are two common ways to achieve this conversion:

**1. Using StringBuilder Constructor:**

You can create a new StringBuilder object by passing the String you want to convert as an argument to the StringBuilder constructor.

***String str = "Hello, World!";***

***StringBuilder sb = new StringBuilder(str); // Convert String to StringBuilder***

**2. Using StringBuilder append() Method:**

Alternatively, you can create an empty StringBuilder object and then use the append() method to append the String content to the StringBuilder.

***String str = "Hello, World!";***

***StringBuilder sb = new StringBuilder();***

***sb.append(str); // Append String content to StringBuilder***

**Section: Array**

**Que: What is an Array in Java?**

**Ans:**

* An array is the collection of similar data types stored in contiguous memory location.
* At the time of declaration of Array, you must specify the type of data with the array name.
* There is the variable associated with the array which refer to the first element in the array.
* You can access different elements present in ana array using their index.
* An array is fixed in length i.e. static in nature.
* An array can hold primitive types and object references.
* Array indexing starts from 0 not 1.

**Que: Mention some advantages and disadvantages of array?**

**Ans:**

**Advantages:**

* **Efficient Access:** Array offer constant time (O (1)) Complexity to access any element by index.
* **Sequential Storage:** As its stored in contiguous memory, so its efficient for sequential access or iterations.
* **Simple and Fast to use.**
* **Static Memory allocation:** Array have fixed size which is determined at compile time which is advantageous when size is known.
* **Compatibility:** Data Structure like Stack and Queue can be efficiently implemented.

**Disadvantages:**

* **Fixed size:** array has fixed size which need to be decide at compile time.
* **Static Memory Allocation:** If not used fully the memory may be wasted.
* **Inflexible:** Once an array is created its size cannot be changed.
* **No Built-in Methods:** InJavaArray doesn’t provide built-in method for common operations like adding or removing elements.
* **Inefficient Insertion and Deletion.**
* **No Support for Heterogeneous Data:** Can store only similar data elements.

**Que: What will happen if you do not initialize an Array?**

**Ans:** The array will take default values depending upon the data type.

**Que: What is the default value of Array in Java?**

**Ans**: If we don't specify the values by ourselves, then Java assigns default values in them which are 0 for byte, short, int, and long, 0.0 for float and double, false for boolean, and null for objects respectively.

**Que: Can you declare an array without assigning the size of an array?**

**Ans:** No, we cannot declare an array without assigning size. If we declare an array without size, it will throw compile time error.

**Que: Difference between Array and Objects?**

**Ans:**

**Array:**

* An array is a fixed size, ordered collection of elements of the same data type. Elements in an array are accessed using integer indices.
* Arrays are declared using square brackets []. They can be initialized with a fixed size and elements of the same data type.

***int[] array = new int[5];***

* Arrays are typically used when dealing with collections of homogeneous data elements, such as a list of numbers, characters, or objects of the same type.

**Object:**

* An object is an instance of a class. It can contain multiple data fields (also known as instance variables) of different data types and methods to manipulate those fields.
* Objects are created using the new keyword followed by a constructor invocation. Objects can be initialized with specific values according to the constructor parameters.

***MyClass obj = new MyClass();***

* Objects are used to represent more complex data structures and entities with multiple attributes and behaviors. They are fundamental to object-oriented programming and allow for abstraction, encapsulation, inheritance, and polymorphism.

**Que: Where is an Array stored in JVM memory?**

**Ans:** An Array is an object in java. So, Array is stored in heap memory in Java Virtual Machine.

**Que: Can a Negative number be passed in Array size?**

**Ans:** No, a negative number cannot be passed as array size. If you pass a negative number in Array size then you will get the NegativeArraySizeException at run time.

**Que: When will we get ArrayStoreException?**

**Ans**: ArrayStoreException is a runtime exception.

For example, you will get this exception at run time if you declare a String Array and then try to insert integer elements in the array.

**Que: When will we get ArrayIndexOutOfBounds Exception?**

**Ans**: ArrayIndexOutOfBounds is a runtime exception that occurs when the program tries to access the invalid index of an array such as an Index higher than the size of the array or a negative index.

**Que: We know that Arrays are objects so why cannot we write strArray.length()?**

**Ans:** We cannot write strArray.length() because length is not a method, it's a data item of an array. We can use the methods of Object like toString() and hashCode() against Array.

In Java, arrays are indeed objects, but they don't have a length() method. Instead, they have a length field. This field is accessed directly without parentheses.

***String[] strArray = {"apple", "banana", "orange"};***

***int arrayLength = strArray.length; // accessing the length field***

***System.out.println("Length of strArray: " + arrayLength);***

Here, strArray.length directly gives you the length of the array without the need for parentheses. This is a property of arrays in Java.

**Que: What is the difference between length and length () in Java?**

**Ans:** In Java, the length() is a method of String class whereas length is an instance variable of an array.

**Length in Java**

* The length variable returns the length of an array i.e. a number of elements present in an array.
* After initializing, the length of an array cannot be changed, so the length variable can directly be used to get the length of an array.
* It is used only for an array.

***int[] array = new int[5];***

***int arrayLength = array.length; // accessing the length property of the array***

**Length() in Java**

* It is a static method of String class.
* The length() returns the number of characters stored in a string object.
* The string class uses this method as the length of a string can be modified using the various operations performed on a string object.
* The String class uses a char[] array internally.

***String str = "Hello, World!";***

***int stringLength = str.length(); // calling the length() method of the String class***

**Que: What is Multi-Dimensional array in Java?**

**Ans:** A multidimensional array in Java is an array that contains other arrays as its elements, forming a nested array structure. These arrays can be of any dimensionality, meaning they can have multiple levels of nesting.

For example, a two-dimensional array is an array of arrays, and a three-dimensional array is an array of arrays of arrays, and so on.

**Que: What is a Jagged Array in Java?**

**Ans:** In Java, a jagged array (also known as a ragged array) is an array of arrays where each row of the array can have a different length. Unlike a regular 2D array, where all rows have the same number of columns, in a jagged array, each row can have a different number of columns.

As an example, we can make a 2D array where the first array contains three elements, and the second array consists of four elements.

**jaggedArray[0] = new int[] {1, 2, 3};**

**jaggedArray[1] = new int[] {4, 5, 6, 7};**

**jaggedArray[2] = new int[] {8, 9};**

**Que: Can the sizeof operator be used to tell the size of an array passed to a function?**

**Ans:** In Java, there is no sizeof operator like there is in languages such as C or C++. Java does not provide a direct way to determine the size of an array, whether it's passed to a function or used elsewhere in the code.

However, you can use the length property of an array in Java to determine its size. This property provides the number of elements in the array.

**Que: What do you mean by the terms “Dimension” and “Subscript” when we talk about arrays?**

**Ans:** In the context of arrays:

**Dimension:** A dimension refers to the number of indices or levels of nesting in an array.

For example, a one-dimensional array has a single dimension, a two-dimensional array has two dimensions, a three-dimensional array has three dimensions, and so on.

**Subscript:** A subscript refers to the index used to access an element within an array. Each dimension of an array has its own subscript or index.

For Example, In a one-dimensional array, there is a single subscript used to access elements. In a two-dimensional array, there are two subscripts: one for the row index and one for the column index. In general, for an N-dimensional array, there are N subscripts corresponding to each dimension.

**Que: Compare Array and Linkedlist?**

**Ans:** Arrays and linked lists are both data structures used to store collections of elements, but they have different characteristics and are suited for different use cases.

|  |  |  |
| --- | --- | --- |
| **Characteristics:** | **Array** | **LinkedList** |
| **Memory Allocation** | Arrays: Contiguous block of memory allocated to hold elements of the array. This means that all elements of the array are stored in adjacent memory locations. | Linked Lists: Elements are stored in nodes, where each node contains the data and a reference (or link) to the next node in the sequence. Nodes can be scattered throughout memory, and they are connected through these references. |
| **Size** | Arrays: Fixed size in most languages (though dynamic arrays like ArrayList in Java exist). Once created, the size cannot be changed without creating a new array. | Linked Lists: Dynamic size. Elements can be added or removed from the list without needing to reallocate memory. |
| **Insertion & Deletion:** | Arrays: Insertion and deletion operations can be expensive, especially if done in the middle of the array, as it may require shifting elements (O(n) time complexity). However, insertion and deletion at the end of the array are efficient (O(1)). | Linked Lists: Insertion and deletion are efficient (O(1) time complexity) as they involve updating references. However, if the position of insertion/deletion is not known and needs to be found, the operation becomes O(n). |
| **Traversal:** | Arrays: Traversal is straightforward using loops, and random access is possible. However, arrays are not suitable for efficient insertion or deletion in the middle of the collection. | Linked Lists: Traversal typically requires iteration through each node in the list, making it less efficient for random access. However, linked lists are well-suited for efficient insertion and deletion, especially in the middle of the list. |

**Que: Why is the complexity of fetching from an Array be O(1)?**

**Ans:** In an Array, objects are stored in continuous memory location. So, if you know the address of the base object then you will be able to find the address of the ith object.

**Que: On which memory arrays are created in Java?**

**Ans:** Arrays in Java are created in heap memory. When an array is created with the help of a new keyword, memory is allocated in the heap to store the elements of the array. In Java, the heap memory is managed by the Java Virtual Machine (JVM).

**Que: Why does the Java array index start with 0?**

**Ans:** The index of an array signifies the distance from the start of the array. So, the first element has 0 distance therefore the starting index is 0.

**Que: What is the difference between int array[] and int[] array?**

**Ans:** Both int array[] and int[] array are used to declare an array of integers in java. The only difference between them is on their syntax no functionality difference is present between them.

**Que: How to copy an array in Java?**

**Ans:** In Java there are multiple ways to copy an Array based on the requirements.

**clone() method in Java:** This method in Java is used to create a shallow copy of the given array which means that the new array will share the same memory as the original array.

***int[] Arr = { 1, 2, 3, 5, 0};***

***int[] tempArr = Arr.clone();***

**arraycopy() method:** To create a deep copy of the array we can use this method which creates a new array with the same values as the original array.

***int[] Arr = {1, 2, 7, 9, 8};***

***int[] tempArr = new int[Arr.length];***

***System.arraycopy(Arr, 0, tempArr, 0, Arr.length);***

*System.arraycopy(Source array Object, int start position in source array, destination array object, int starting position od destination array,int length of elements to be copied to new)*

**copyOf() method:** This method is used to create a new array with a specific length and copies the contents of the original array to the new array.

***int[] Arr = {1, 2, 4, 8};***

***int[] tempArr = Arrays.copyOf(Arr, Arr.length);***

Arrays.copyOf(Original Array obj, New array length)

**copyOfRange() method:** This method is very similar to the copyOf() method in Java, but this method also allows us to specify the range of the elements to copy from the original array.

***int[] Arr = {1, 2, 4, 8};***

***int[] temArr = Arrays.copyOfRange(Arr, 0, Arr.length);***

***arrays.copyofRange(original array, int from index, int to index )***

**Que: Is it possible to make an array volatile?**

**Ans:** In Java, it is not possible to make a volatile. Volatile keywords in Java can only be applied to individual variables but not to arrays or collections.

The **volatile keyword** is used to indicate that a variable's value may be changed by multiple threads concurrently.

However, it's important to note that arrays themselves cannot be declared as volatile.

When you declare an array as volatile, you are actually making the reference to the array volatile, not the elements within the array. This means that changes to the reference itself (such as assigning a new array instance to the reference) will be visible to other threads immediately, but changes to the elements of the array may not be visible immediately to other threads.

**Object oriented Programming In Java**

**Que: What is Object oriented Programming in Java?**

**Ans:** OOPs refers to Object-Oriented Programming. It is the programming paradigm that is defined using objects. Objects can be considered as real-world instances of entities like class, that have some characteristics and behaviors.

The term “characteristics” refers to the “what” about the Object, and the term “behavior” refers to the “how” about the Object.

For example, if we consider a car, then based on the OOPs model:

* Class = A specific car model, such as Audi A4, BMW I8, Maruti Suzuki Vitara Brezza, etc.
* Object = A specific car of any model, like the car you own
* Characteristics = What is the color of your car? What is the Chassis number of your car?
* Behavior = How to start the car? How to change the gear of the car? etc.

**Que: Why do we need OOPs?**

**Que: What are some advantages of using OOPs?**

**Que: Why is OOPs so Popular ?**

**Ans:** OOPs is a paradigm that organizes software design around objects and data rather that actions and logic.

Several reasons to have OOPs are:

1. Modularity: By encapsulating related attributes into objects.

2. Reusability: Oops promote code reuse by inheritance.

3. Abstraction: You can hide your implementation.

4. Encapsulation: Bundle data and methods within a single unit.

5. Polymorphism: extensibility of code.

6. Ease of Maintenance: Oops promotes code organization.

7. Code Understandability: OOPs encourages a natural modeling of real-world entities so easy to understand.

**Que: What is the difference between an object-oriented programming language and an object-based programming language?**

**Ans:**

| **Feature** | **OOPL** | **OBPL** |
| --- | --- | --- |
| Inheritance | Core feature, supports class hierarchies | Limited or no support for inheritance |
| Polymorphism | Core feature, supports method overriding | May have limited support or different implementation |
| Encapsulation | Core feature, bundles data and methods | May or may not be fully supported |
| Examples | Java, C++, Python, C# | JavaScript (to some extent), Visual Basic, MATLAB |

**Que: What are the four main Pillar of OOPs?**

**Ans:** Object Oriented Programming mainly comprises of the below four features, and make sure you don't miss any of these:

1. Inheritance
2. Encapsulation
3. Polymorphism
4. Data Abstraction

**Class and Objects and then OOPs**

**Que: What are Classes in Java?**

**Ans:** In Java, Classes are the collection of objects sharing similar characteristics and attributes.

A class represents the blueprint or template from which object are created.

Classes are not real-world entities but help us to create real-world entities.

So, when an object is created, it automatically takes the data and functions that are defined in the class.

For Example, first, a car’s template is created. Then multiple units of car are created based on that template.

**Que: What is an object?**

**Ans:**  An object refers to the instance of a class. The object is a real-world entity that has certain properties and method associated with it. An object is created using New Keyword.

**Que: What is New Keyword in Java?**

**Que: How is the ‘new’ operator different from the ‘newInstance()’ operator in Java?**

**Ans:** In Java, the new operator is used to create a new instance of a class by calling its constructor.

For example:

***MyClass obj = new MyClass();***

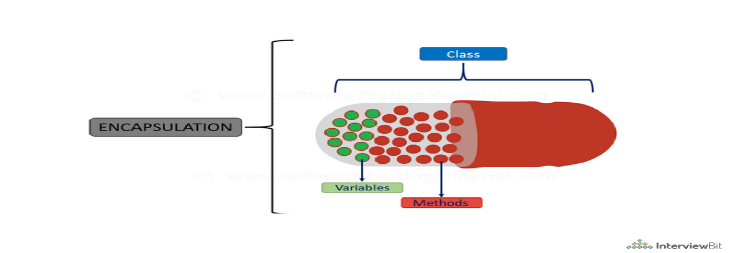
* On the other hand, newInstance() is a method provided by the Class class in Java, which can be used to create new instances of classes dynamically at runtime.
* This method is used to invoke a zero-argument constructor of a class represented by the Class object.

***Class<?> clazz = MyClass.class;***

***MyClass obj = (MyClass) clazz.newInstance();***

**Que: What is Encapsulation?**

**Ans:**



Encapsulation is the process of binding data members and methods of a program together to do a specific job, without revealing unnecessary details.

One can visualize Encapsulation as the method of putting everything that is required to do the job, inside a capsule and presenting that capsule to the user. All the necessary data and methods are bind together and all the unnecessary details are hidden to the normal user.

Encapsulation can also be defined in two different ways:

**1) Data Hiding:** Encapsulation is the process of hiding unwanted information, such as restricting access to any member of an object.

**2) Data Binding**: Encapsulation is the process of binding the data members and the methods together as a whole, as a class.

E.g... In a capsule all the medicinal properties are added. All the salt and their properties are combined that is encapsulated in a capsule, and those properties of salt is hidden from the user , the user cannot know that what is inside that capsule , because it is irrelevant for that user.

**Que: How Encapsulation is Achieved in Java?**

**Ans:** in Java is achieved by combining data (variables) and methods that operate on the data into a single unit, typically a class.

Here are steps:

**1. Declare Instance Variable as Private**

* Declare instance variables (fields) as private to restrict direct access from outside the class.

***public class MyClass {***

***private int myVariable;***

***}***

2**. Provide Public Getter and Setter Methods:**

* Provide public getter methods to access the values of private instance variables.
* Provide public setter methods to modify the values of private instance variables.
* This allows controlled access to the object's state and enforces encapsulation by encapsulating the data within the class.

***public class MyClass {***

***private int myVariable;***

***// Getter method***

***public int getMyVariable() {***

***return myVariable;***

***}***

***// Setter method***

***public void setMyVariable(int value) {***

***myVariable = value;***

***}***

***}***

**3. Implement Validation and Logic Inside Setter Methods (Optional):**

* Inside setter methods, implement validation checks and additional logic to ensure data integrity.
* This allows you to enforce constraints and maintain consistency in the object's state.

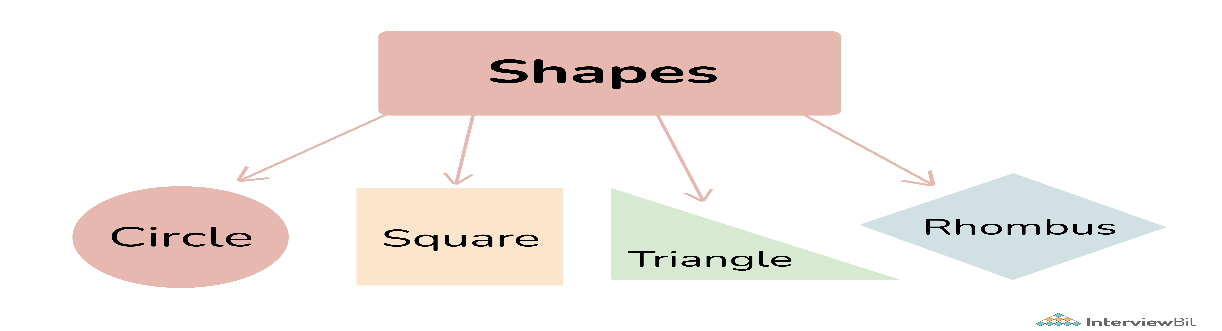
**Que: What are the advantages of Encapsulation in Java?**

**Ans**: The advantages of Encapsulation in Java are mentioned below:

1. Data Hiding: it is a way of restricting the access of our data members by hiding the implementation details. Encapsulation also provides a way for data hiding. The user will have no idea about the inner implementation of the class.
2. Increased Flexibility: We can make the variables of the class read-only or write-only depending on our requirements.
3. Reusability: Encapsulation also improves the re-usability and is easy to change with new requirements.
4. Testing code is easy: Code is made easy to test for unit testing.

**Que: What is Polymorphism?**

**Ans:** Polymorphism is composed of two words - “poly” which means “many”, and “morph” which means “shapes”. Therefore, Polymorphism refers to something that has many shapes.



Polymorphism refers to the process by which some code, data, method, or object behaves differently under different circumstances or contexts.

Example: Consider a person, who can have multiple characteristics at a time, the person can be a father, a son, and an employee at the same time.

***public class Calculator {***

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

***return a + b;***

***}***

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

***return a + b;***

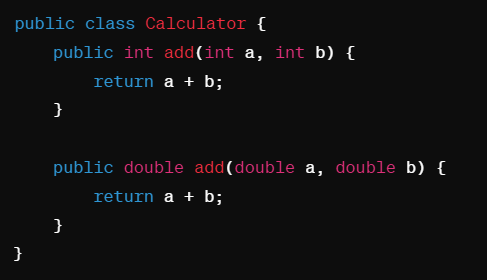
***}***

***}***

**Que: How Polymorphism is Achieved in Java?  
Ans:** Polymorphism in Java can be achieved through two main mechanisms: method overloading and method overriding. Here's how each mechanism works:

**1. Method Overloading:**

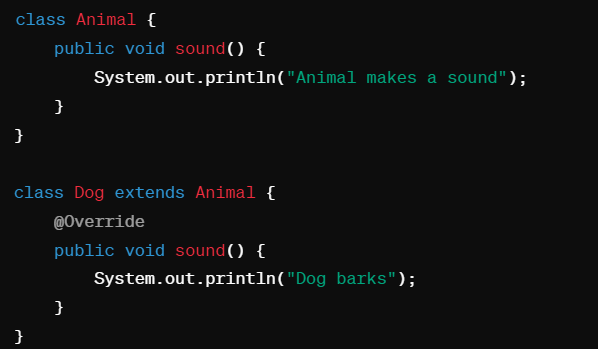
Method overloading allows a class to have multiple methods with the same name but different parameter lists. The methods must have different parameter types, number of parameters, or order of parameters.



In the example above, the add method is overloaded with two versions, one that accepts integers and another that accepts doubles. The appropriate method is called based on the arguments provided at compile-time.

**2. Method Overriding:**

Method overriding allows a subclass to provide a specific implementation of a method that is already defined in its superclass. This enables the subclass to provide its own behavior for the method, thus achieving run-time polymorphism.

****

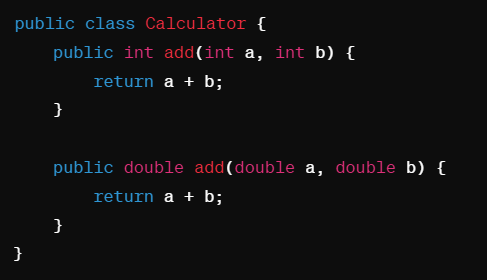
In this example, the Dog class overrides the sound method of its superclass Animal. When calling sound on a Dog object, the overridden method in the Dog class is executed instead of the one in the Animal class.

**Que: What are the types of Polymorphism in Java?**

**Ans:** There are two main types of polymorphism in Java:

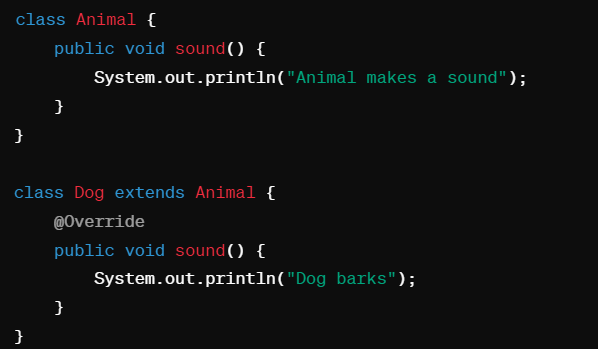
**1. Compile-time Polymorphism (Method Overloading):**

* Compile-time polymorphism, also known as method overloading, occurs when there are multiple methods in a class with the same name but different parameters.
* The compiler determines which method to call based on the number and type of arguments passed to it at compile time.
* This is Also known as Static or early Binding.



**2.Run-time Polymorphism (Method Overriding):**

* Run-time polymorphism, also known as method overriding, occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.
* The method invoked is determined at runtime based on the actual type of the object.
* This is also known as Dynamic or Late Binding.

****

**Que: What are benefits of Polymorphism?**

**Ans:** The benefits of polymorphism in Java summarized in short points:

1. Code Reusability: Enables methods to work with objects of different classes, promoting reuse.
2. Flexibility and Extensibility: Facilitates adding new subclasses without modifying existing code.
3. Enhanced Readability and Maintainability: Promotes cleaner, more readable code by treating objects at a higher level of abstraction.
4. Dynamic Method Binding: Determines the method to be invoked at runtime based on the object's type.
5. Improved Design Patterns Implementation: Supports various design patterns that enhance code structure and flexibility.

**Que: What is Dynamic Method Dispatch?**

**Ans:** In Java, dynamic method dispatch is associated with method overriding, which allows a subclass to provide a specific implementation of a method that is already defined in its superclass.

Object is of sub class and reference of super class. But vice versa not allows.

Super s= new sub(); true

Sub s= new super(); false

Eg: a smart to is also tv but a TFT tv is not a smart tv.

Methods are called depending on the object not depending on the reference.

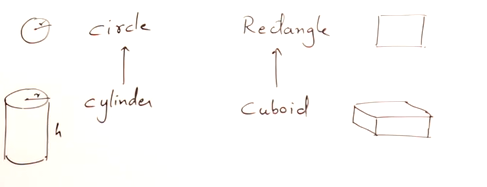
Super class holding and sub class object🡪 overridden method will be called.

Dispatch means calling . dynamic means dynamically calling a method.

**Que: What is meant by Inheritance in Java?**

**Ans:** Inheritance is the process of acquiring the features of existing class to a new class, i.e.. borrowing the features.

The term “inheritance” means “receiving some quality or behavior from a parent to an offspring.” In object-oriented programming, inheritance is the mechanism by which an object or class (referred to as a child) is created using the definition of another object or class (referred to as a parent).



Circle is super class and cylinder is sub class.

Cylinder have all the properties of circle but have an extra properties i.e.. Height.

Cylinder is inherited the properties of circle.

Everything from circle is **available** for cylinder.

**Que: What are the types of Inheritance in Java?**

**Ans:** Inheritance can be categorized into several types based on the relationship between the classes involved and the structure of the inheritance hierarchy.

**1. Single Inheritance:**

* In single inheritance, a subclass inherits from only one superclass.
* Java supports single inheritance, meaning that a class can extend only one superclass.
* Example:

***class SubClass extends SuperClass {***

***// Subclass members***

***}***

**2. Multilevel Inheritance:**

* In multilevel inheritance, a subclass inherits from another subclass, forming a chain of inheritance.
* This creates a hierarchical inheritance structure where each subclass extends the superclass of the previous level.
* Example:

***class SubClass extends SuperClass {***

***// Subclass members***

***}***

***class SubSubClass extends SubClass {***

***// Subsubclass members***

***}***

**3. Hierarchical Inheritance:**

* In hierarchical inheritance, multiple subclasses inherit from a single superclass.
* This creates a tree-like structure where a superclass serves as the root, and multiple subclasses extend from it.
* Example:

***class SubClass1 extends SuperClass {***

***// Subclass1 members***

***}***

***class SubClass2 extends SuperClass {***

***// Subclass2 members***

***}***

**4. Multiple Inheritance (Through Interfaces):**

* Java does not support multiple inheritance of classes (where a class extends more than one superclass) to avoid the diamond problem.
* However, Java supports multiple inheritance through interfaces, where a class can implement multiple interfaces.
* This allows a class to inherit behavior (method signatures) from multiple interfaces while maintaining a single superclass.
* Example:

***interface Interface1 {***

***// Interface1 methods***

***}***

***interface Interface2 {***

***// Interface2 methods***

***}***

***class MyClass implements Interface1, Interface2 {***

***// Class members***

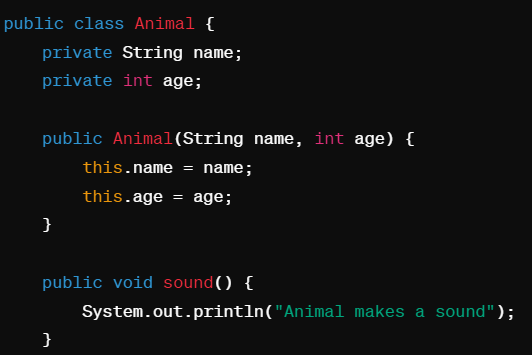
***}***

**Que: How Inheritance is achieved in Java?**

Ans: In Java, inheritance is achieved using the extends keyword. Here's how to implement inheritance:

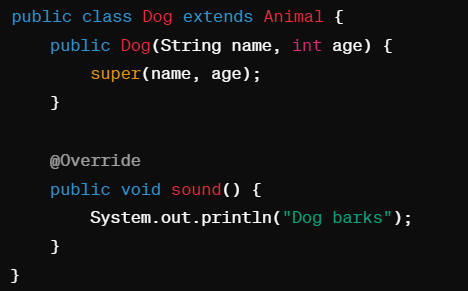
**1. Define the Superclass:**

* Start by defining a superclass (also called a base class or parent class). This class contains common attributes and behaviors that you want to share among multiple subclasses.



**2. Define the Subclass:**

* Create a subclass (also called a derived class or child class) that extends the superclass. The subclass inherits all the fields and methods of the superclass.

****

**3. Accessing Superclass Members:**

* Within the subclass, you can access the superclass members (fields and methods) using the super keyword.

**4. Constructor Invocation:**

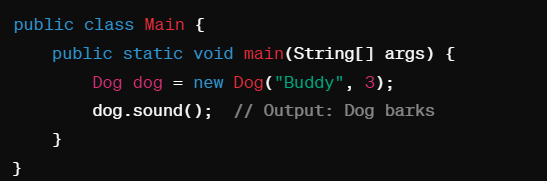
* If the superclass has a constructor, the subclass constructor should call it using the super() keyword.
* This initializes the inherited members of the superclass.

**4. Method Overriding:**

* Optionally, you can override methods of the superclass in the subclass to provide specialized behavior.
* Use the @Override annotation to indicate that a method is being overridden.

**5. Creating Objects:**

* Create objects of the subclass and access the inherited members as well as the members defined in the subclass.



**Que: Benefits of Inheritance in Java?**

Ans: Here are the benefits of inheritance in Java summarized in short points:

* Code Reusability: Inherit fields and methods from a superclass, reducing duplication.
* Extensibility: Subclasses can add new functionality or override existing behavior.
* Modularity: Organizes classes into hierarchical structures, enhancing code organization.
* Polymorphism: Enables objects of different classes to be treated uniformly.
* Reduced Complexity: Allows for specialization, managing complexity effectively.
* Facilitates Design Patterns: Forms the foundation for various design patterns, enhancing code structure and flexibility.

**Que: What are limitations of Inheritance?**

**Ans:** Here are the limitations of inheritance in Java summarized in short points:

* **Inflexibility and Tight Coupling**: Can lead to rigid class relationships and tight dependencies.
* **Fragile Base Class Problem:** Changes in superclass can break functionality in subclasses.
* **Hierarchical Constraints**: Complex hierarchies can be difficult to manage and maintain.
* **Limited to Single Inheritance:** A class can extend only one superclass, limiting flexibility.
* **Semantic Coupling:** Subclasses may inherit unnecessary or irrelevant behavior.
* **Violation of Encapsulation:** Can expose internal details of superclass, violating encapsulation.
* **Difficulty in Testing and Debugging:** Complex hierarchies can make testing and debugging challenging.

**Que: What is a subclass?**

**Ans:** The subclass is a part of Inheritance. The subclass is an entity, which inherits from another class. It is also known as the child class.

**Que: Define a superclass?**

**Ans:** Superclass is also a part of Inheritance. The superclass is an entity, which allows subclasses or child classes to inherit from itself.

**Que: What is Abstraction in Java?**

**Ans:** Abstraction in Java involves hiding the complex implementation details of a class and exposing only the essential features of an object. It allows programmers to focus on what an object does rather than how it does it. Abstraction is achieved through abstract classes and interfaces in Java.

For example, consider a car. You only need to know how to run a car, and not how the wires are connected inside it. This is obtained using Abstraction.

**Que: How to achieve Abstraction?**

Ans: Abstraction is achieved through abstract classes and interfaces in Java.

**1. Abstract Classes:**

* An abstract class is a class that cannot be instantiated on its own and may contain abstract methods.
* Abstract methods are declared without implementation, leaving it to subclasses to provide concrete implementations.
* Abstract classes may also contain non-abstract methods with implemented behavior.
* Abstract classes are declared using the abstract keyword.

**2. Interfaces:**

* An interface is a reference type in Java that defines a contract of methods that a class implementing the interface must implement.
* Interfaces can contain only abstract methods (in Java 7 and earlier) or a combination of abstract methods and default/static methods (in Java 8 and later).
* Classes can implement multiple interfaces, allowing for multiple inheritance of type.

**Que: What are Benefits of Abstraction:**

**Ans:**

* **Encapsulation of Complexity:** Hides internal details of implementation, providing a simplified view of objects.
* **Modularity**: Promotes separation of concerns by defining clear interfaces between components.
* **Code** **Reusability**: Allows for the creation of reusable components through interfaces and abstract classes.
* **Flexibility and Extensibility:** Enables polymorphic behavior, allowing objects to be treated uniformly based on their abstraction.

**Que: What are Abstract Classes?**

**Ans:** In Java, an abstract class is a class that cannot be instantiated on its own and may contain both abstract and non-abstract methods. Abstract classes are primarily used as blueprints for other classes to inherit from. They serve as partial implementations of a class, allowing subclasses to provide concrete implementations for abstract methods while inheriting non-abstract methods and fields.

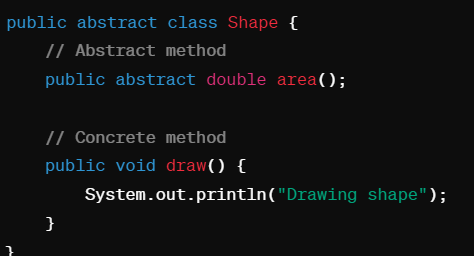
**Key characteristics of abstract classes in Java:**

**1. Abstract Methods:**

* Abstract classes can contain abstract methods, which are declared without an implementation.
* Abstract methods are meant to be overridden by subclasses, providing specific implementations.

**2. Concrete Methods:**

* Abstract classes may also contain concrete methods with implementations.
* Subclasses inherit these concrete methods along with the abstract methods and fields of the abstract class.

****

**3. Instantiation Restriction:**

* Abstract classes cannot be instantiated directly; you cannot create objects of abstract classes.
* They can only be used as superclasses for other classes to inherit from.

**4. Inheritance:**

* Subclasses of an abstract class must either provide implementations for all abstract methods or be declared abstract themselves.

**5. Abstract Keyword:**

* Abstract classes are declared using the abstract keyword in their class declaration.

**Que: Characteristics of Abstract Class?**

**Ans:**

* Abstract method: Method doesn’t have body.
* Abstract void method2(); 🡪 and semi colon after method.
* Must be declared as abstract.
* Abstract method means undefined method. When nothing is defined.
* Abstract class is that class that contains zero or more abstract method.
* Abstract class can have zero abstract method by just defining the class as abstract.
* When something is not defined then we can’t create the object.
* When a class is abstract then object.
* If a class inherited from abstract class then the sub class will also be abstract.
* To make that sub class concrete class you must Override the method now the object can be create of sub class.
* Abstract class and method cannot be final.
* Abstract class and method cannot be static.
* Abstract class is useful for polymorphism.

**Example of Abstract Class:**

Maine apna frenchisi lia KFC ka lekin usko log KFC hi bolange mtlb ki we are using the reference of KFC.

BILLING is abstraxt bcz billing mera hoga offer mere yaha ka hoga.

Note: all the method are by default abstract and public.

**Que: Why is Abstract Class Needed?**

**Ans:** Abstract Class play a vital role in Java.

1. **Abstraction**: Abstract classes define abstract concepts without implementation details.

2. **Code Reusability**: Contains both abstract and concrete methods, promoting code reuse.

3. **Enforcing** **Method** **Signatures**: Abstract methods ensure subclasses provide specific behavior.

4. **Polymorphism**: Allows objects of different subclasses to be treated uniformly.

5. **Encapsulation** of **Common** **Behavior**: Centralizes shared functionality for better code organization.

6. **Framework** **Design**: Commonly used to design frameworks, providing structure and behavior.

7. **Modeling** **Real**-**world** **Entities**: Useful for modeling entities with common behavior but varying implementations.

Abstract class is used for defining standard and imposing standards.

**Que: When Abstract methods are used?**

**Ans:** An abstract method is used when we want to use a method but want to child classes to decide the implementation in that case we use Abstract methods with the parent classes.

**Que: What are the limitations of Abstract Class ?**

**Ans:** The limitations of abstract classes in Java presented as concise points:

**1. Partial Abstraction:** Combining abstract and concrete methods limits full abstraction**.**

**2. Single Inheritance:** Restricted to inheriting from only one abstract class.

**3. Constructor Invocation:** Abstract classes' constructors can't be directly invoked.

**4. Semantic Coupling:** May lead to subclasses inheriting unnecessary behavior.

**5. Tight Coupling:** Subclasses are tightly bound to the superclass, impacting flexibility.

**6. Complexity:** Large hierarchies can introduce complexity, affecting code readability.

**7. Single Root:** Abstract classes must have a single root, restricting hierarchy design.classes must have a single root, restricting hierarchy design.

**Que: What are Interfaces in Java?**

**Ans: Interfaces:**

* Interface is a abstract class with all abstract methods.
* Its is implemented.
* We can achieve polymorphism using interfaces.
* In java a class can extends from only one class, but an interface can implement from more than one class.
* We can have reference of interface but we cannot create object.
* Same hi hota hai abstract class jaisa … standard maintain karne ke lie.
* When a class is implementing a interface it is also a abstract class until you override the method.
* Interface are only meant for achieving override so to achieve run-time polymorphism s so on to dynamic method dispatch.

An interface in Java is like a blueprint or a contract that defines a set of methods that a class must implement. It declares what methods a class should have without specifying how those methods are implemented.

**1. Declaration:**

* Interfaces are declared using the interface keyword in Java.
* They contain method signatures but no method implementations.

**2. Method Signatures:**

* Interfaces declare method signatures without specifying the method body.
* Methods in interfaces are by default implicitly public and abstract.
* Java 8 introduced default methods and static methods in interfaces, allowing for method implementation within the interface itself.

**3. Multiple Inheritance:**

* Unlike classes, a class in Java can implement multiple interfaces.
* This feature allows Java to achieve a form of multiple inheritance through interfaces.

**4. Implementation by Classes:**

* Classes that implement an interface must provide concrete implementations for all the methods declared in the interface.
* Implementing classes must use the implements keyword to declare that they implement a specific interface.

**5. Polymorphism:**

* Interfaces enable polymorphic behavior, allowing objects of different classes to be treated uniformly based on the interface they implement.
* Interface references can be used to refer to objects of implementing classes.

**6. Framework Design:**

* Interfaces are widely used in framework design to define a contract or API that classes must adhere to.
* They provide a way to define a set of operations that classes must support without specifying how those operations are implemented.

**Que: What are the types of Interfaces in Java?**

**Ans:** In Java, interfaces can be categorized into different types based on their purpose and the methods they contain. Here are the main types of interfaces:

**1. Regular Interfaces:**

* Regular interfaces are the most common type of interface in Java.
* They declare a set of abstract methods that classes can implement.
* Prior to Java 8, interfaces could only contain abstract method declarations.

**2. Functional Interfaces:**

* Introduced in Java 8, functional interfaces are interfaces that contain exactly one abstract method (SAM - Single Abstract Method).
* They can have any number of default or static methods, but only one abstract method.
* Functional interfaces are commonly used with lambda expressions and method references to enable functional-style programming in Java.

**3. Marker Interfaces:**

* Marker interfaces are interfaces that do not declare any methods.
* They serve as markers or tags to provide metadata about classes that implement them.
* Examples include Serializable, Cloneable, and Random Access in Java.

**4. Default Methods:**

* Introduced in Java 8, default methods allow interfaces to provide default implementations for methods.
* Classes that implement the interface can choose to override the default method or use the default implementation.
* Default methods provide a way to add new methods to existing interfaces without breaking backward compatibility.

**5. Static Methods:**

* Also introduced in Java 8, static methods allow interfaces to define static methods that can be called without an instance of the interface.
* Static methods in interfaces are commonly used as utility methods related to the interface.

**Que: What is Functional interface and how to create Functional interface ?**

**Ans:** A functional interface in Java is an interface that contains only one abstract method. Functional interfaces are used to represent lambda expressions or method references, allowing you to treat functions as objects. Java 8 introduced functional interfaces as part of the lambda expressions feature. They are annotated with the @FunctionalInterface annotation, although the annotation is optional.

**1. Define the Interface:**

* Start by defining an interface with a single abstract method. This method represents the functionality you want to encapsulate.
* Use the @FunctionalInterface annotation to explicitly mark the interface as functional (optional but recommended).

***@FunctionalInterface***

***public interface MyFunctionalInterface {***

***void myMethod(); // Single abstract method***

***}***

**2.Implement the Functional Interface:**

* Once you have defined the functional interface, you can implement it using lambda expressions or method references.

***public class Main {***

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

***// Using lambda expression to implement MyFunctionalInterface***

***MyFunctionalInterface funcInterface = () -> System.out.println("Executing myMethod");***

***funcInterface.myMethod(); // Output: Executing myMethod***

***}***

***}***

* You can also implement the functional interface using an anonymous inner class, but lambda expressions provide a more concise syntax.

***public class Main {***

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

***// Using anonymous inner class to implement MyFunctionalInterface***

***MyFunctionalInterface funcInterface = new MyFunctionalInterface() {***

***@Override***

***public void myMethod() {***

***System.out.println("Executing myMethod");***

***}***

***};***

***funcInterface.myMethod(); // Output: Executing myMethod***

***}***

***}***

**3. Using Method References (Optional):**

* Instead of using lambda expressions, you can also implement a functional interface using method references.

***public class Main {***

***public static void myMethodImplementation() {***

***System.out.println("Executing myMethodImplementation");***

***}***

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

***// Using method reference to implement MyFunctionalInterface***

***MyFunctionalInterface funcInterface = Main::myMethodImplementation;***

***funcInterface.myMethod(); // Output: Executing myMethodImplementation***

***}***

***}***

**Que: What is SAM?**

**Ans**: In the context of Java programming, SAM stands for "Single Abstract Method." A SAM type refers to an interface that contains exactly one abstract method. SAM types are often used in Java to enable functional programming features, such as lambda expressions and method references.

Here are a few key points about SAM types:

**1. Functional Interface**: A SAM type is essentially a functional interface, which means it can be used with lambda expressions and method references.

**2. Lambda Expressions**: When you use a lambda expression to represent an instance of a SAM type, you are essentially providing an implementation for the single abstract method defined in that interface.

**3. Method References:** SAM types can also be implemented using method references, which provide a shorthand syntax for referring to existing methods.

**Que: What are the advantages of Functional Interface?**

Ans: Functional interfaces in Java offer several advantages that facilitate the development of more expressive, concise, and maintainable code, especially when combined with lambda expressions and the Stream API. Here are some of the key advantages of using functional interfaces:

**Enables Functional Programming:**

Functional interfaces are fundamental to functional programming in Java. They allow you to treat functions as first-class citizens, enabling you to pass functions as arguments, return functions from methods, and assign functions to variables.

**Lambda Expressions and Method References:**

Functional interfaces are designed to be used with lambda expressions and method references. They provide a clear and concise way to represent behavior as data, leading to more expressive and readable code.

**Encourages Modular and Reusable Code:**

By defining behavior in functional interfaces, you can create modular and reusable components. This promotes code organization and reduces duplication, leading to more maintainable and scalable applications.

**Facilitates Stream API and Data Processing:**

Functional interfaces are integral to the Stream API introduced in Java 8. They enable functional-style operations on collections and streams, such as mapping, filtering, reducing, and collecting. This leads to more efficient and declarative data processing code.

**Promotes Separation of Concerns:**

Functional interfaces help in separating the definition of behavior from its implementation. This separation of concerns allows you to focus on what needs to be done (the behavior) without worrying about how it's done (the implementation details).

**Flexibility and Customization:**

Functional interfaces provide flexibility in defining custom behavior for specific use cases. You can create your own functional interfaces tailored to your application's requirements, allowing for customized and extensible solutions.

**Parallelism and Concurrency:**

Functional interfaces combined with parallel streams and CompletableFuture enable parallelism and concurrency in Java applications. They make it easier to write concurrent code by providing a functional approach to handle parallel tasks.

**Improved Error Handling:**

Functional interfaces can be used to define functional-style error handling using interfaces like Supplier, Consumer, Function, etc. This leads to cleaner and more structured error handling mechanisms.

**Que: Name some Functional Interface in Java ?**

**Ans**: In Java, several functional interfaces are part of the Java API and are commonly used in functional programming paradigms, particularly with lambda expressions and method references. Here are some of the frequently used functional interfaces:

**1.Runnable:**

* Represents a task that can be executed asynchronously.
* Contains a single abstract method void run().

***Runnable task = () -> System.out.println("Executing task");***

**2. Callable:**

* Similar to Runnable but can return a result and throw a checked exception.
* Contains a single abstract method V call() throws Exception.

***Callable<Integer> task = () -> 42;***

**3. Supplier:**

* Represents a supplier of results.
* Contains a single abstract method T get().

***Supplier<String> supplier = () -> "Hello, World!";***

**4. Consumer:**

* Represents an operation that accepts a single input argument and returns no result.
* Contains a single abstract method void accept(T t).
* Used to perform side-effects.

***Consumer<String> printUC= str->System.out.print(str.toUppercase);***

***printUC.accept("hello");***

**5. Function:**

* Represents a function that accepts one argument and produces a result.
* Contains a single abstract method R apply(T t).
* It transforms the elements.
* Example in maps.

***Function<Integer, String> converter = (num) -> "Converted: " + num;***

**6. Predicate**:

* Represents a predicate (boolean-valued function) of one argument.
* Contains a single abstract method boolean test(T t).
* Example used in Filter.
* It returns true and false.

***Predicate<Integer> isEven = (num) -> num % 2 == 0;***

**7. BiFunction:**

* Represents a function that accepts two arguments and produces a result.
* Contains a single abstract method R apply(T t, U u).

***BiFunction<Integer, Integer, Integer> sum = (a, b) -> a + b;***

**8. UnaryOperator:**

* Represents an operation on a single operand that produces a result of the same type as its operand.
* Extends Function<T, T> and contains a single abstract method T apply(T t).

***UnaryOperator<Integer> square = (num) -> num \* num;***

**Que: What are Marker Interfaces?**

**Ans:** Marker interfaces, also known as tag interfaces, are a type of interface in Java that does not declare any methods. Instead, they serve as markers or flags to indicate certain properties or capabilities of a class. The presence of a marker interface on a class allows the Java runtime or other tools to treat the class differently or enable specific functionalities.

Here are some key points about marker interfaces in Java:

**1. No Methods**: Marker interfaces do not contain any methods. They are essentially empty interfaces with no method definitions.

***// Example of a marker interface***

***public interface Serializable {***

***// No methods defined***

***}***

**2. Signifying Capabilities**: Marker interfaces are used to signify that a class implementing the interface has certain capabilities, properties, or requirements. This information can be used by the Java runtime or other tools.

**3. Examples of Marker Interfaces:**

* java.io.Serializable: Indicates that a class can be serialized and deserialized using Java's serialization mechanism.
* java.lang.Cloneable: Indicates that a class supports the clone() method for creating a copy of an object.
* java.util.RandomAccess: Indicates that a list implementation supports fast random access to elements.

**Que: What are Lambda Expression? Give some of its examples and Why we use it? What are some of the advantages of Lambda Expression?**

**Ans:**

* A Lambda Expression is a concise way to represent an anonymous function.
* It doesn't have a name, but it can have parameters, a body, and a return type.
* It allows you to treat functionality as a method argument, creating more compact and expressive code.
* Lambda expressions were introduced in Java 8 as part of the functional programming enhancements.
* Here's the general syntax of a lambda expression:

***(parameters) -> expression or statement block***

* (parameters) -> expression or statement block
* Parameters: Zero or more parameters that the lambda expression can take.
* Arrow (->): Separates the parameters from the body of the lambda expression.
* Expression or Statement Block: Represents the functionality of the lambda expression.

Example of Lambda Expression:

***List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "David");***

***names.sort((s1, s2) -> s1.compareTo(s2));***

***System.out.println(names); // Output: [Alice, Bob, Charlie, David]***

**Advantages of Lambda Expressions:**

**1. Conciseness:** Lambda expressions allow you to write more compact code compared to traditional anonymous classes or method implementations.

**2. Expressiveness:** Lambda expressions make your code more expressive by focusing on the actual functionality rather than the mechanics of defining a method or class. This can lead to clearer and more understandable code.

**3. Functional Programming:** Lambda expressions promote functional programming concepts by treating functions as first-class citizens. This enables you to use functional interfaces, streams, and other functional programming constructs more effectively.

**4. Code Flexibility:** Lambda expressions provide flexibility in defining inline behavior, making it easier to pass behavior as an argument to methods or use it in functional programming paradigms.

**5. Parallelism:** Lambda expressions can be used with parallel streams to take advantage of multi-core processors and improve performance by parallelizing operations.

**6. Readability**: Lambda expressions, when used appropriately, can enhance the readability of your code by focusing on the intent of the functionality rather than the implementation details.

**Que: Is lambda expression actually overriding the abstract method of functional interface ?**

**Ans:** Yes, that's correct! When you use a lambda expression to create an instance of a functional interface, you are essentially providing an implementation for the single abstract method defined in that interface. Lambda expressions allow you to override the abstract method of a functional interface in a concise and expressive manner.

***@FunctionalInterface***

***interface MyFunctionalInterface {***

***void myMethod();***

***}***

***MyFunctionalInterface funcInterface = () -> System.out.println("Executing myMethod");***

**Java 8**

**Que: What are all features of Java 8?**

**Ans:** Java 8 introduced several significant features and enhancements to the Java programming language. Here are the key features introduced in Java 8:

**1. Lambda Expressions:**

Lambda expressions allow you to treat functionality as a method argument, enabling a more concise and expressive way to represent anonymous functions.

Syntax: (parameters) -> expression or statement block.

**2. Functional Interfaces:**

Functional interfaces are interfaces that contain exactly one abstract method. They are used in conjunction with lambda expressions and method references.

**3. Stream API:**

The Stream API provides a functional-style approach to process collections of objects. It supports operations such as map, filter, reduce, and collect.

**4. Default and Static Methods in Interfaces:**

Interfaces can now have default and static methods, allowing for method implementations within interfaces.

**5. Optional Class:**

The Optional class provides a more robust way to handle null values and avoid NullPointerExceptions.

**6. Date and Time API:**

The new Date and Time API (java.time package) provides improved date and time handling with classes like LocalDate, LocalTime, LocalDateTime, ZonedDateTime, etc.

**7. Default Methods in Interfaces:**

Default methods allow you to provide default implementations for interface methods. This feature helps in evolving interfaces without breaking existing implementations.

**8. Method References:**

Method references provide a shorthand syntax for lambda expressions, making it easier to refer to existing methods.

**9. Parallel Streams:**

Parallel streams allow for parallel processing of data using multiple threads, enhancing performance for certain operations.

**10. CompletableFuture:**

The CompletableFuture class provides a way to write asynchronous, non-blocking code using a fluent API.

CompletableFuture.supplyAsync(() -> "Hello")

.thenApply(s -> s + " World")

.thenAccept(System.out::println);

**Que: What is Default and Static method in Java 8?**

**Ans**: In Java 8, default and static methods were introduced in interfaces as a way to provide method implementations directly within interfaces without requiring implementing classes to provide their own implementations.

**1. Default Methods:**

* A default method in an interface is a method that provides a default implementation. It is defined using the default keyword.
* Default methods allow interfaces to evolve by adding new methods without breaking existing implementations of those interfaces.
* Implementing classes can choose to override default methods if needed.

***interface Vehicle {***

***default void start() {***

***System.out.println("Vehicle started");***

***}***

***}***

***class Car implements Vehicle {***

***// No need to override defaultMethod***

***}***

**2. Static Methods:**

* A static method in an interface is a method that is associated with the interface itself, rather than with any instance of the interface or implementing classes. It is defined using the static keyword.
* Static methods in interfaces are primarily used as utility methods or helper methods.

***interface MathUtils {***

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

***return a + b;***

***}***

***}***

***class MyClass {***

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

***int sum = MathUtils.add(5, 3); // Calling static method from interface***

***System.out.println("Sum: " + sum);***

***}***

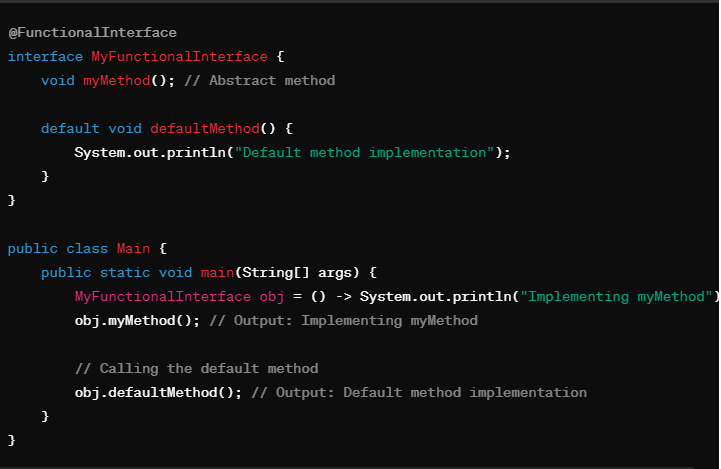
***}***

**Que: What are Default Methods in functional interface?**

**Ans:** In Java, a functional interface is an interface that contains exactly one abstract method. Java 8 introduced the concept of default methods in interfaces, which are methods that provide a default implementation directly within the interface.

**Default Methods in Functional Interfaces:**

* A functional interface can have one or more default methods in addition to the single abstract method required for functional interfaces.
* Default methods in a functional interface provide a default implementation for methods other than the abstract method.
* Default methods allow functional interfaces to add new methods without forcing existing implementations to provide implementations for these new methods.
* Default methods are defined using the default keyword.



In this example:

* MyFunctionalInterface is a functional interface that contains a single abstract method myMethod() and a default method defaultMethod().
* The main method demonstrates creating an instance of MyFunctionalInterface using a lambda expression to implement myMethod().
* The default method defaultMethod() is also called on the instance of MyFunctionalInterface.

**Que: What is the need of Default method in Functional Interface?**

**Ans:** Default methods in functional interfaces serve several important purposes:

**1.Backward Compatibility:**

* Default methods allow for the addition of new methods to existing interfaces without breaking backward compatibility with implementing classes.

**2. Interface Evolution:**

* Default methods enable interface evolution by allowing interfaces to evolve over time without disrupting existing code.

**3. Code Reusability:**

* Default methods promote code reusability by providing default implementations for common functionalities in interfaces.

**4. Utility Methods:**

* Default methods can be used to define utility methods or helper methods within interfaces.
* These utility methods are related to the functionality of the interface and can be accessed by implementing classes.

**Que: Explain Method Referencing in Java 8 ?**

**Ans:** Method references in Java 8 provide a shorthand syntax for lambda expressions that simply call an existing method. They allow you to refer to methods without explicitly defining a lambda expression with the method's parameters and body. Method references are especially useful when you want to pass a method as an argument to a higher-order function or when you want to simplify lambda expressions that only call a single method.

There are four types of method references in Java 8:

**1.Reference to a Static Method:**

Syntax: ClassName::staticMethodName

// Lambda expression

***Function<String, Integer> parseInt = s -> Integer.parseInt(s);***

// Method reference

***Function<String, Integer> parseIntRef = Integer::parseInt;***

**2. Reference to an Instance Method of a Particular Object:**

Syntax: object::instanceMethodName

// Lambda expression

***Consumer<String> print = s -> System.out.println(s);***

// Method reference

***Consumer<String> printRef = System.out::println;***

**3. Reference to an Instance Method of an Arbitrary Object of a Particular Type**:

Syntax: ClassName::instanceMethodName

***List<String> list = Arrays.asList("apple", "banana", "cherry");***

// Lambda expression

***list.forEach(s -> System.out.println(s.toUpperCase()));***

// Method reference

***list.forEach(String::toUpperCase);***

**4. Reference to a Constructor:**

Syntax: ClassName::new

// Lambda expression

***Supplier<List<String>> listSupplier = () -> new ArrayList<>();***

// Method reference

***Supplier<List<String>> listSupplierRef = ArrayList::new;***

**Que: What is Date and Time API in Java 8? Was it present in earlier version of Java? What's new introduced in Java 8?**

**Ans:** The Date and Time API in Java 8 refers to the java.time package, which introduced new classes and methods for handling date, time, and related concepts. Prior to Java 8, Java had the java.util.Date and java.util.Calendar classes for date and time operations, but these classes had several limitations and were not well-suited for modern date and time programming needs.

The Date and Time API in Java 8 addressed many of these limitations and introduced several new features:

**1. Key Classes:**

**LocalDate**: Represents a date without a time component (e.g., "2022-04-12").

**LocalTime**: Represents a time without a date component (e.g., "10:30:00").

**LocalDateTime**: Represents a date and time without a time zone (e.g., "2022-04-12T10:30:00").

**ZonedDateTime**: Represents a date and time with a time zone (e.g., "2022-04-12T10:30:00+01:00 Europe/London").

**Instant**: Represents a point in time on the time-line in UTC (Coordinated Universal Time).

**2. Immutable and Thread-Safe:**

The new Date and Time API classes are immutable and thread-safe, which improves concurrency and avoids common pitfalls with mutable date/time objects.

**3.Improved Date/Time Manipulation:**

The API provides comprehensive methods for date/time manipulation, including arithmetic operations, parsing/formatting, duration calculations, and more.

**Constructor**

**Que: What are Constructors in Java?**

**Ans:** In Java, A Constructor is a special type of method that is automatically called when an object of a class is created. Constructors are used to initialize the state of an object by providing initial values to its instance variables or performing other necessary setup tasks. Here are the key points about constructors in Java:

**Syntax:**

* A constructor has the same name as the class and does not specify a return type, not even void.
* It may or may not have parameters.

**Initialization:**

* Constructors are primarily used to initialize the state of an object by setting initial values to its instance variables.
* Initialization tasks such as allocating memory, acquiring resources, or performing validation checks can also be performed in a constructor.

**Automatic Invocation:**

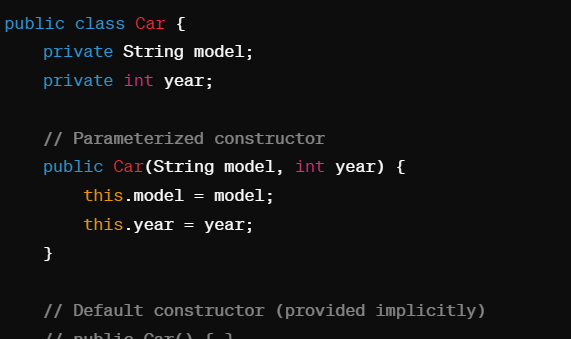
* Constructors are automatically called when an object of a class is created using the new keyword.
* They are invoked implicitly by the JVM and executed before the object is made available for use.

**Default Constructor:**

* If a class does not explicitly define any constructors, Java provides a default constructor with no parameters.
* The default constructor initializes instance variables to their default values (0, null, or false).

**Overloading:**

* Like other methods, constructors can be overloaded by defining multiple constructors with different parameter lists.
* Overloaded constructors provide flexibility in creating objects with different initialization requirements.

****

**Summary:**

* Whenever an object is called a constructor is created by default.
* Every class in java has a default constructor. It created by compiler.
* If you want to use your you have to create.
* Constructor is a method of a class which will have same name as class.
* Constructor will be called automatically while creating an object.
* Constructor will not have any return type. Don’t even write void.
* Constructor as usually public but can be private.

**Que: What are types of constructors in Java?**

**Ans:** In Java, constructors can be categorized into different types based on their parameters, usage, and invocation.

**1. Default Constructor:**

* A default constructor is automatically provided by Java if no constructor is explicitly defined in a class.
* It has no parameters and initializes instance variables to their default values (0, null, or false).

**2. Parameterized Constructor:**

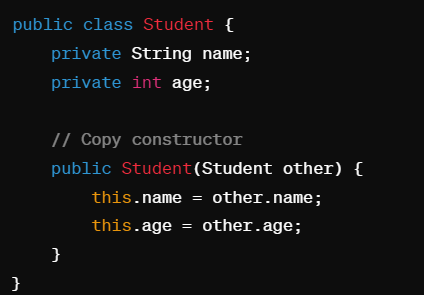
* A parameterized constructor is a constructor that takes one or more parameters.
* It allows for custom initialization of instance variables based on the provided parameters.

**3. Non-Parameterized Constructor:**

* A parameterized constructor is a constructor that takes no parameters.

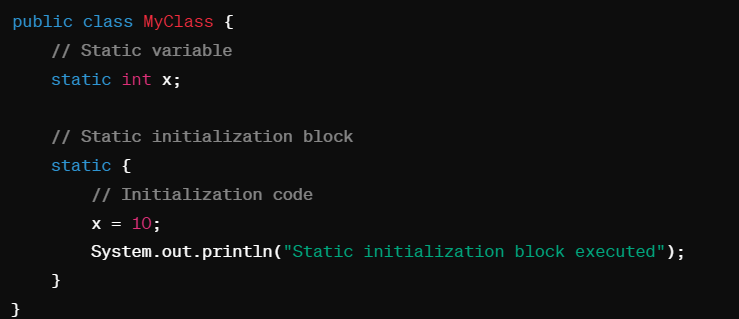
**4. Copy Constructor:**

* A copy constructor is a constructor that takes an object of the same class as a parameter.
* It initializes a new object by copying the state of the existing object.



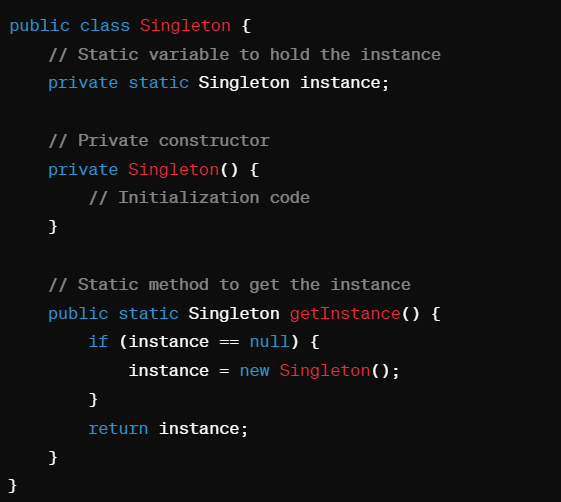
**5. Static Constructor (Initializer Block):**

* Static initialization blocks, also known as static constructors, are used to initialize static variables in a class.
* They are executed only once when the class is loaded into memory.



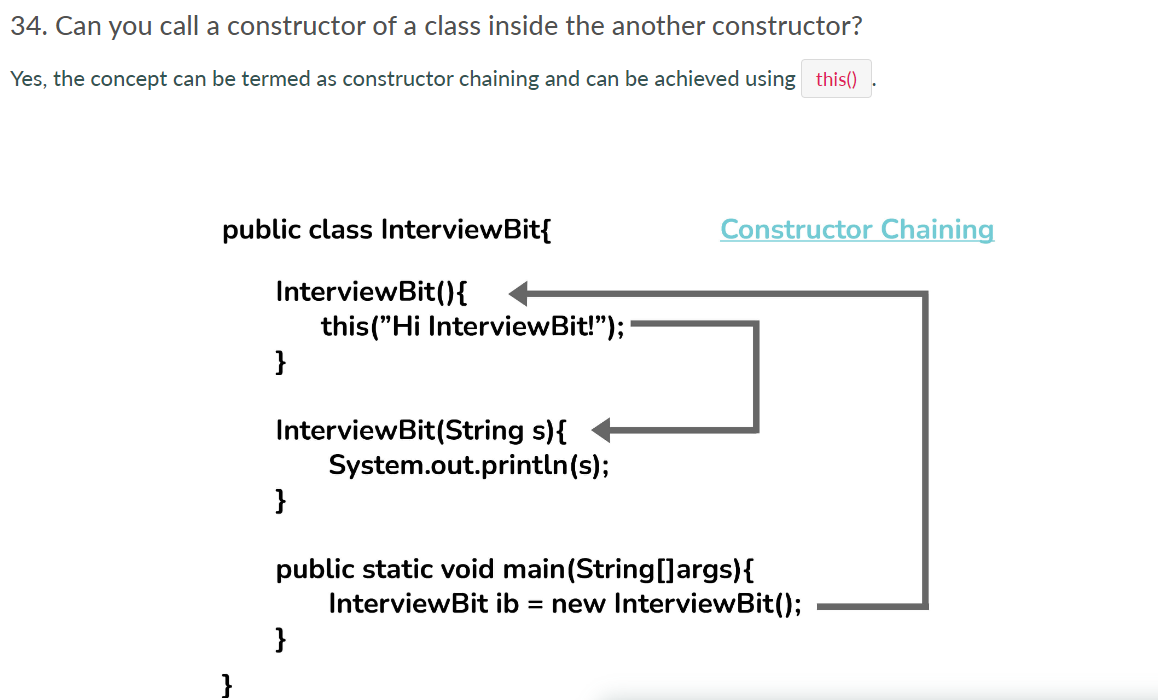
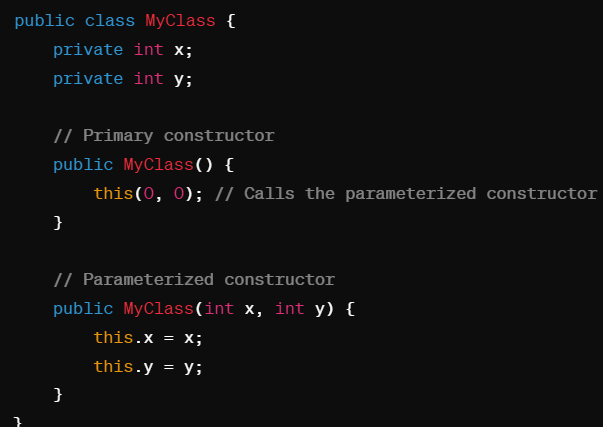
**6. Private Constructor:**

* A private constructor is a constructor that is declared with the private access modifier.
* It restricts the instantiation of a class from outside the class itself, typically used in singleton design pattern implementations.



**7. Chain Constructor:**

* A chain constructor is a constructor that invokes another constructor in the same class using the this() keyword.
* It allows for constructor chaining, where one constructor calls another to perform common initialization tasks.



**8. Constructor Overloading:**

* Constructor overloading involves defining multiple constructors within the same class with different parameter lists.
* It provides flexibility in creating objects with different initialization requirements.

**Que: What is the need of constructor in Java?**

Ans: The key points summarizing the need for constructors in Java:

* **Initialization**: Constructors initialize the state of an object.
* **Object Creation:** Automatically invoked when an object is created.
* **Encapsulation:** Helps encapsulate initialization logic within the class.
* **Customization**: Allows multiple constructors for object creation customization.
* **Inheritance:** Ensures proper initialization in subclass objects.
* **Resource Allocation**: Can be used for resource allocation and management.
* **Initialization Order:** Specifies the order of initialization tasks.

**Que: What are the limitations of constructors?**

Ans: The limitations of constructors in Java summarized in short points:

* **Non-Inheritance**: Constructors are not inherited by subclasses, each subclass must define its own constructors.
* **No Return Type:** Constructors cannot return values, not even void.
* **Cannot be Overridden:** Constructors cannot be overridden in subclasses.
* **Limited Access:** Private constructors restrict subclass inheritance and external instantiation.
* **Name Restriction**: Constructors must have the same name as the class they belong to.
* **No Constructor Inheritance**: Subclasses cannot directly access superclass constructors.
* **No Synchronization or Native:** Constructors cannot be declared as synchronized or native.

**Exception Handling interview Question**

**Que: What is an Exception in java?**

**Ans:** An exception is an abnormal event that disrupts the flow of normal program execution that unless handled could lead to the termination of that program. In real-world scenarios, a program could run into an exception if it tries to access a corrupted/non-existent file or if a network error happens or if the JVM runs out of memory, or if the code is trying to access an index of an array which does not exist and so on.

**Que: Difference between Error and Exception?**

**Ans:**

**Error**:

* Errors are abnormal conditions that typically arise due to severe issues in the Java Virtual Machine (JVM) or the underlying system.
* They usually indicate serious problems that cannot be recovered or handled by the application code. Examples of errors include OutOfMemoryError, StackOverflowError, and NoClassDefFoundError.
* Errors are typically caused by problems in the runtime environment or system-level issues.
* They are often beyond the control of the application code and require intervention at the system level or by adjusting JVM settings.

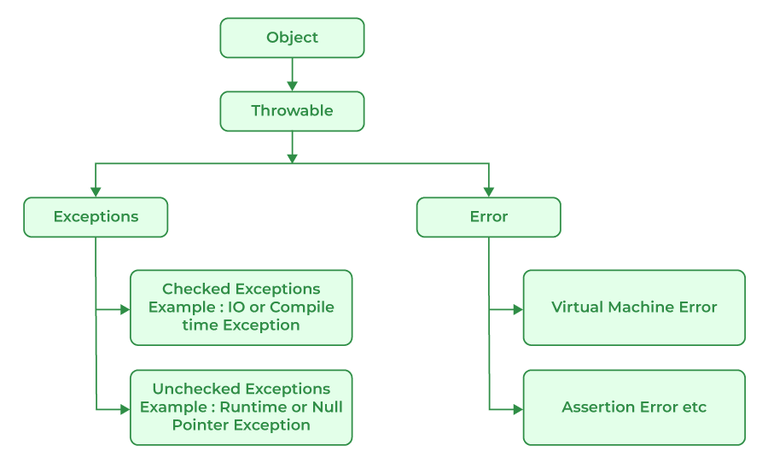
**Exception**:

* Exceptions, on the other hand, are abnormal conditions or events that occur during the execution of a program but can be handled and recovered from.
* Exceptions can be caused by various factors such as invalid input, file I/O errors, network issues, arithmetic errors, etc.

**Que: State the Hierarchy of Exception in Java?**

**Ans:** All exception and error types are subclasses of the class Throwable, which is the base class of the hierarchy. One branch is headed by Exception. This class is used for exceptional conditions that user programs should catch.

Another branch, Error is used by the Java run-time system(JVM) to indicate errors having to do with the run-time environment itself(JRE).



**Que: What is Exception Handling in Java?**

**Ans:** Exception handling in Java is a mechanism that allows developers to manage and respond to runtime errors or exceptional conditions that may occur during program execution. These exceptions can be caused by various factors such as invalid inputs, file not found, network issues, arithmetic errors, and so on. By handling exceptions properly, developers can ensure that their programs can gracefully recover from errors and continue running without crashing.

**Que: What are advantages of Exception Handling in Java?**

**Ans:** Exception handling in Java offers several advantages that contribute to writing robust, reliable, and maintainable code:

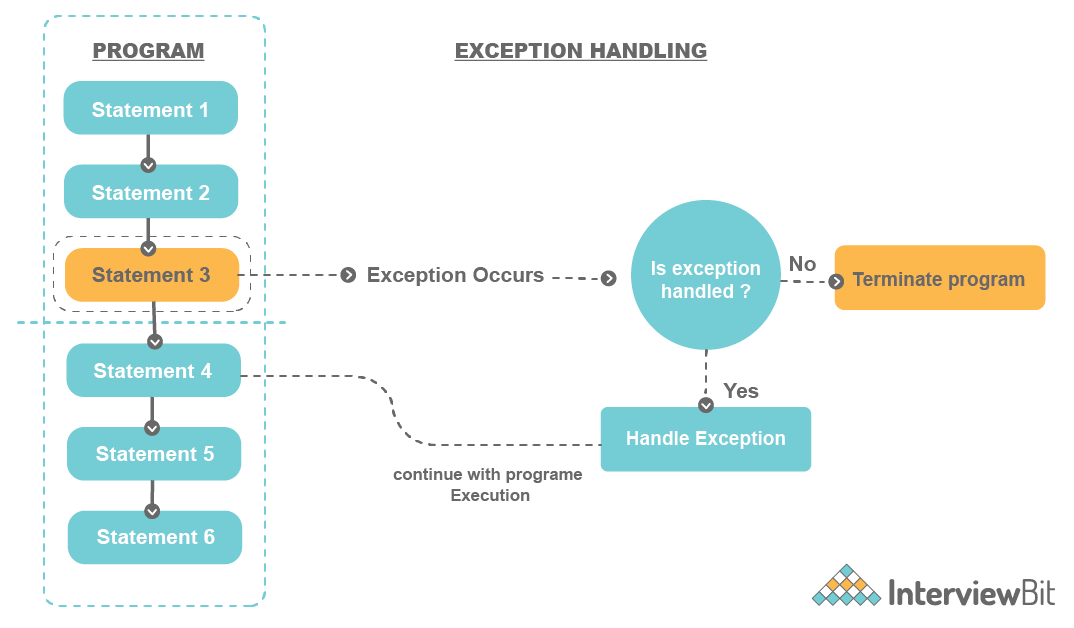
**1. Error Detection:** Exception handling helps in detecting and identifying errors or exceptional conditions during program execution. Instead of crashing the program abruptly, exceptions provide a mechanism to gracefully handle errors and recover from them.

**2. Graceful Recovery:** Exception handling allows developers to implement strategies for recovering from errors and continuing program execution. This helps in maintaining the overall stability of the application and providing a better user experience by handling errors gracefully.

**3. Graceful Recovery:** Exception handling allows developers to implement strategies for recovering from errors and continuing program execution. This helps in maintaining the overall stability of the application and providing a better user experience by handling errors gracefully.

**4. Debugging and Logging:** Exception handling facilitates debugging and troubleshooting processes by providing detailed information about the cause of errors. Java exceptions contain stack traces that help developers identify the exact location and sequence of code where an exception occurred.

**5. Custom Exception Types:** Java allows developers to create custom exception classes, enabling them to define and handle application-specific exceptional conditions. Custom exceptions improve code clarity and help communicate specific error scenarios effectively.



**Que: How are exceptions handled in Java?**

**Ans:** Exceptions are handled in Java using a combination of try-catch blocks, finally blocks, and optional throw statements. Here's how exceptions are handled in Java:

**1. Try-Catch Blocks:**

* The try block is used to enclose the code that may throw an exception.
* The catch block is used to handle specific types of exceptions that may occur within the try block.
* You can have multiple catch blocks to handle different types of exceptions.

***try {***

***// Code that may throw an exception***

***} catch (ExceptionType1 e1) {***

***// Handle ExceptionType1***

***} catch (ExceptionType2 e2) {***

***// Handle ExceptionType2***

***} finally {***

***// Optional finally block for cleanup operations***

***}***

**2. Finally Block:**

* The finally block is used to execute code that should always be run regardless of whether an exception occurs or not.
* It is often used for cleanup operations such as closing open resources (files, database connections, etc.).
* The finally block is optional, but if used, it follows the catch blocks.
* The syntax for a try-catch-finally block looks like this:

***try {***

***// Code that may throw an exception***

***} catch (ExceptionType e) {***

***// Handle the exception***

***} finally {***

***// Cleanup code (e.g., closing resources)***

***}***

**3. Throwing Exceptions:**

* Developers can explicitly throw exceptions using the throw keyword.
* This is useful when a specific condition is met, and the program needs to indicate an error or exceptional situation.
* The syntax for throwing an exception looks like this:

***if (condition) {***

***throw new ExceptionType("Error message");***

***}***

**4. Throws Method Signature:**

* Java provides the throws keyword is used in method declarations to indicate that the method may throw certain types of exceptions during its execution.
* When a method is declared with the throws clause, it informs the caller that the method may potentially throw exceptions of the specified types, and the caller is responsible for handling those exceptions or propagating them further up the call stack.

**5. Exception Hierarchy:**

* Java provides a hierarchy of exception classes, with java.lang.Throwable as the root class.
* Subclasses of Throwable include Exception (for checked exceptions) and RuntimeException (for unchecked exceptions).
* You can catch exceptions based on their specific types or use broader catch blocks to catch multiple types of exceptions.

**6. Custom Exceptions:**

* Developers can create custom exception classes by extending existing exception classes like Exception or RuntimeException.
* Custom exceptions are helpful for creating more meaningful and specific exception types tailored to the application's needs.
* Example of creating a custom exception:

***public class CustomException extends Exception {***

***public CustomException(String message) {***

***super(message);***

***}***

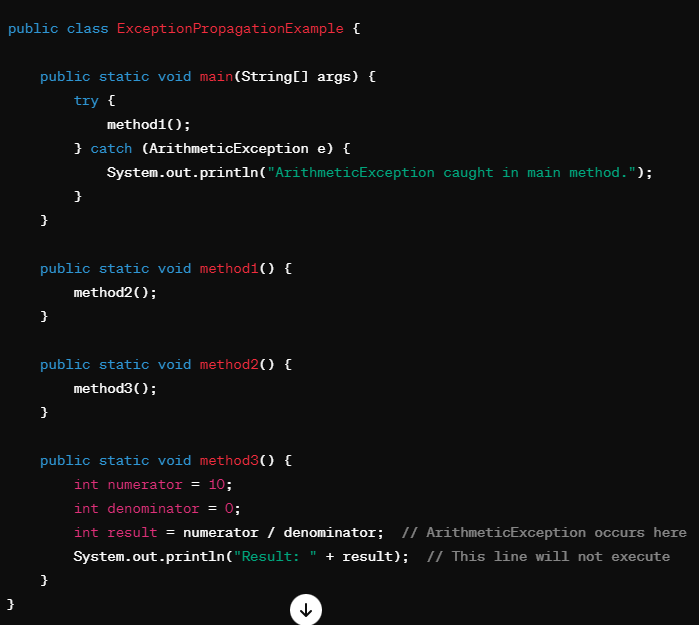
***}***

**Que: What is exception propagation in Java?**

**Ans:** Exception propagation in Java refers to the process by which an exception thrown in a method is passed or propagated up the call stack until it is handled by an appropriate try-catch block or reaches the top level of the program where it may cause the program to terminate if not handled.

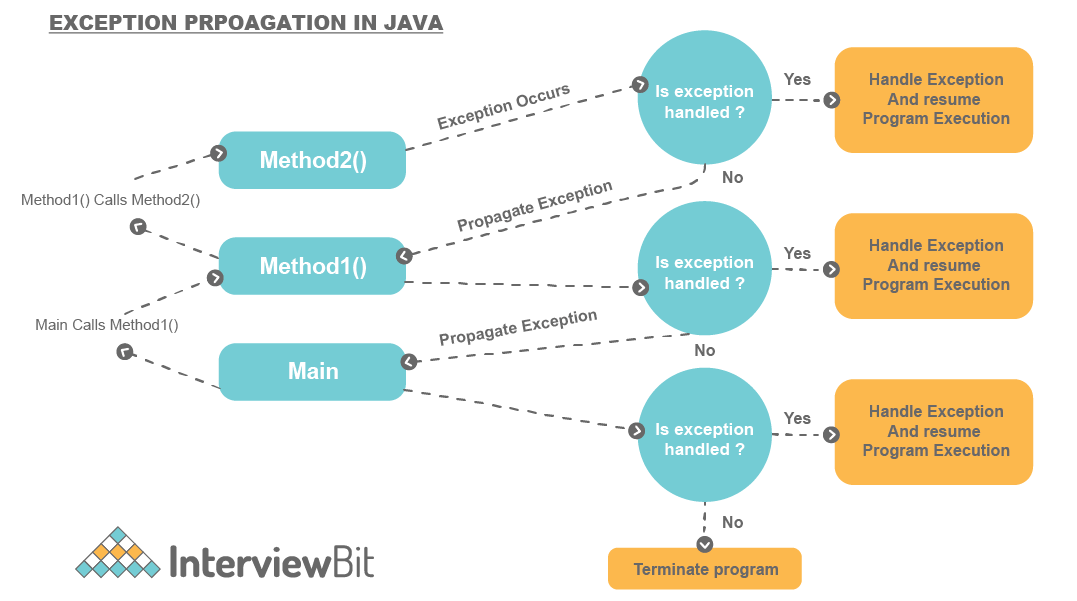
In this example:

1. The main method is the entry point of the program.
2. The main method calls method1 within a try-catch block to handle any potential ArithmeticException.
3. method1 then calls method2, which in turn calls method3.
4. In method3, an arithmetic error occurs when attempting to divide by zero (numerator / denominator), leading to an ArithmeticException.



Here's how exception propagation occurs step by step:

1. The main method calls method1.
2. method1 calls method2.
3. method2 calls method3, where the ArithmeticException occurs due to the division by zero.
4. Since method3 does not handle the exception locally (i.e., it doesn't have a try-catch block), the exception is propagated back to method2.
5. Similarly, since method2 doesn't handle the exception either, the exception is propagated back to method1.
6. Again, since method1 doesn't handle the exception, it continues to propagate up to the main method.
7. In the main method, the exception is caught by the try-catch block that specifically catches ArithmeticException.
8. The catch block in main executes, and the program continues its execution normally after handling the exception.
9. If the exception were not caught in the main method, it would continue to propagate to the JVM's default exception handler, potentially causing the program to terminate and printing the exception's stack trace.



**Que: What are the important methods defined in Java’s Exception Class?**

**Ans:** Here are some of the important methods defined in java.lang.Exception:

**1. getMessage():**

This method is inherited from java.lang.Throwable.

It returns a detailed error message string that describes the cause of the exception.

Example:

**try {**

**// Code that may throw an exception**

**} catch (Exception e) {**

**String errorMessage = e.getMessage();**

**System.out.println("Error Message: " + errorMessage);**

**}**

**2. toString():**

* Also inherited from java.lang.Throwable, this method returns a string representation of the exception object.
* By default, it includes the exception's class name and the message returned by getMessage().

***try {***

***// Code that may throw an exception***

***} catch (Exception e) {***

***String exceptionString = e.toString();***

***System.out.println("Exception String: " + exceptionString);***

***}***

**3. printStackTrace():**

* This method is inherited from java.lang.Throwable.
* It prints the stack trace of the exception to the standard error stream (System.err).
* The stack trace includes the sequence of method calls leading to the exception, helping developers identify where the exception occurred.

***try {***

***// Code that may throw an exception***

***} catch (Exception e) {***

***e.printStackTrace();***

***}***

**4. getCause():**

* Inherited from java.lang.Throwable, this method returns the cause of the exception or null if the cause is unknown or nonexistent.
* It is commonly used in conjunction with chained exceptions (exceptions that are caused by other exceptions).

***try {***

***// Code that may throw an exception***

***} catch (Exception e) {***

***Throwable cause = e.getCause();***

***if (cause != null) {***

***System.out.println("Cause of the Exception: " + cause.toString());***

***}***

***}***

**Que: What are the types of exception?**

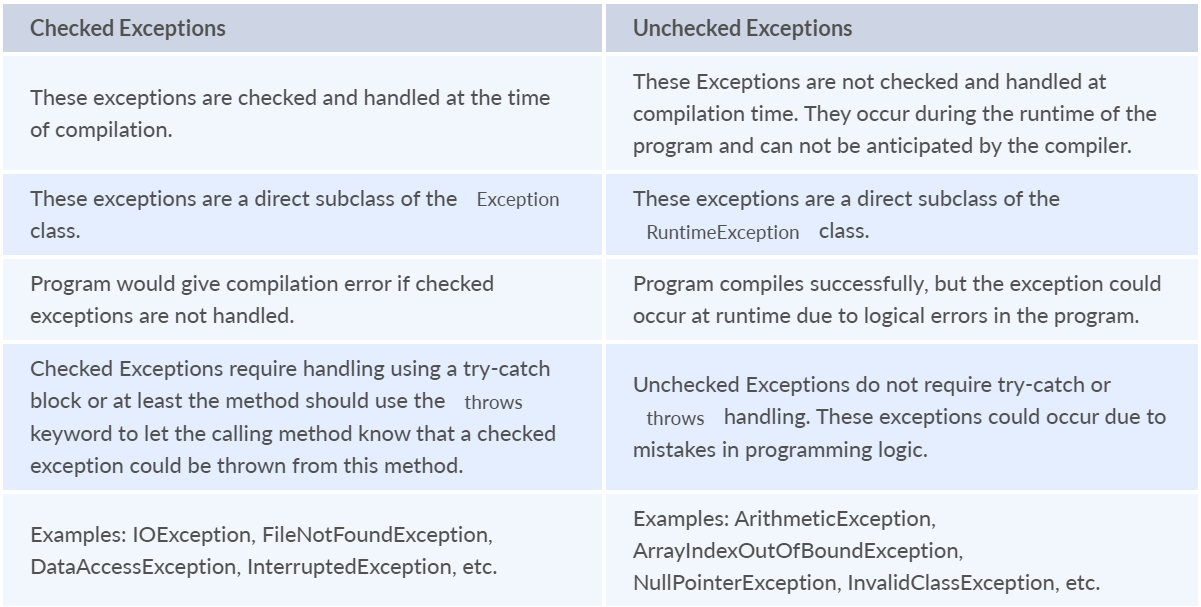
**Ans:** In Java, exceptions are categorized into two main types: checked exceptions and unchecked exceptions. These types are based on whether the compiler enforces handling or declaration of the exception.

**1. Checked Exceptions:**

* Checked exceptions are exceptions that must be either caught (handled) or declared using the throws clause in the method signature.
* These exceptions are subclasses of java.lang.Exception (excluding subclasses of RuntimeException).
* Checked exceptions are typically used for exceptional conditions that can occur during normal program execution and that a well-behaved application should anticipate and handle.
* Examples of checked exceptions include IOException, SQLException, FileNotFoundException, and custom checked exceptions that extend java.lang.Exception.

**2. Unchecked Exceptions (Runtime Exceptions):**

* Unchecked exceptions, also known as runtime exceptions, do not need to be explicitly caught or declared using the throws clause.
* These exceptions are subclasses of java.lang.RuntimeException or its subclasses.
* Unchecked exceptions typically represent programming errors or unexpected conditions that may occur during runtime and are not easily recoverable.
* Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, IllegalArgumentException, ClassCastException, and custom unchecked exceptions that extend java.lang.RuntimeException.

****

**Checked Exception:**

List of some common checked exceptions in Java:

1. IOException
2. SQLException
3. ClassNotFoundException
4. ParseException
5. InterruptedException
6. FileNotFoundException
7. NoSuchMethodException
8. CloneNotSupportedException
9. IllegalAccessException
10. InstantiationException
11. NoSuchFieldException

**Unchecked Exception:**

List of some common unchecked exceptions (runtime exceptions) in Java:

1. NullPointerException
2. ArrayIndexOutOfBoundsException
3. IllegalArgumentException
4. IllegalStateException
5. ClassCastException
6. NumberFormatException
7. UnsupportedOperationException
8. ConcurrentModificationException
9. SecurityException
10. ArithmeticException

**Que: What is the difference between the throw, throws keywords and throwable in Java?**

**Ans:**

**1. throw:**

* The throw keyword is used to explicitly throw an exception within a method.
* It is used when a specific error condition or exceptional situation occurs during the execution of a method, and the program needs to indicate that error by throwing an exception object.
* When you use throw, you provide an instance of an exception class or one of its subclasses to represent the error being thrown.

**2. throws:**

* The throws keyword is used in method declarations to indicate that the method may potentially throw one or more types of exceptions during its execution.
* It is used in the method signature (before the method body) to specify the types of exceptions that the method can throw, allowing the caller to handle or propagate those exceptions.
* When a method declares throws, it means that the method does not handle the specified exceptions internally but instead delegates the responsibility of handling those exceptions to its caller.
* When a method is declared with the throws clause, it informs the caller that it may encounter exceptions of the specified types, and the caller is responsible for handling those exceptions or propagating them further up the call stack.

***public void readFile(String fileName) throws IOException {***

***// Code that may throw IOException***

***}***

* throws hum method me lagate hai aur wo btate hai ki isme exception ho sakta hai aur wo uss exception ko previous method me dal deta hai jaha se wo method call hua tha.

***public class FileReaderExample {***

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

***try {***

***readFile("example.txt");***

***} catch (IOException e) {***

***System.err.println("Error reading file: " + e.getMessage());***

***}***

***}***

***public static void readFile(String fileName) throws IOException {***

***// Method implementation that may throw IOException***

***// Simulating file reading operation (not actual code)***

***if (!fileName.equals("example.txt")) {***

***throw new IOException("File not found");***

***}***

***System.out.println("File content: Sample text from " + fileName);***

***}***

***}***

**3. Throwable Class:**

* The Throwable class is the superclass of all errors and exceptions in Java's exception hierarchy.
* It is the root class for both checked exceptions (java.lang.Exception) and unchecked exceptions (java.lang.RuntimeException and its subclasses).
* Instances of Throwable or its subclasses are used to represent exceptional conditions that occur during program execution.

**Que: Que: What are Try-catch blocks?**

**Ans:** Try-catch blocks are a fundamental part of Java's exception handling mechanism. They are used to manage and handle exceptions that may occur during the execution of a block of code. Here's an explanation of try-catch blocks:

**1. Try Block:**

* The try block is used to enclose the code that may potentially throw an exception.
* It is followed by one or more catch blocks or a finally block.
* Inside the try block, you place the code that you want to monitor for exceptions.
* If an exception occurs within the try block, the control immediately jumps to the corresponding catch block (if applicable) or the finally block.

**2. Catch Block:**

* A catch block follows a try block and is used to handle specific types of exceptions that may be thrown within the try block.
* You can have multiple catch blocks to handle different types of exceptions, each with its own exception type.
* When an exception of a specific type occurs in the try block, the control transfers to the corresponding catch block that matches the type of the thrown exception.
* Inside the catch block, you write code to handle the exception, such as logging an error message or taking corrective actions.

**3. Finally Block:**

* A finally block is optional and follows all catch blocks (if present) in a try-catch structure.
* The code inside the finally block is always executed, regardless of whether an exception occurred or not.
* Typically, finally blocks are used for cleanup operations, such as closing resources (files, database connections, etc.), releasing locks, or freeing up memory.
* Even if an exception is thrown and caught, the code in the finally block will still be executed before the control is transferred out of the try-catch structure.

Example:

***public class TryCatchFinallyExample {***

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

***try {***

***// Code that may throw an exception***

***int result = divideNumbers(10, 0); // Division by zero will throw***

***ArithmeticException***

***System.out.println("Result: " + result); // This line will not execute due***

***to exception.***

***} catch (ArithmeticException e) {***

***// Handle the ArithmeticException***

***System.err.println("Error: Division by zero");***

***} finally {***

***// Finally block for cleanup operations***

***System.out.println("Inside finally block");***

***}***

***// This line will execute after try-catch-***

***finally block***

***System.out.println("Program execution continues after exception handling");***

***}***

***public static int divideNumbers(int numerator, int denominator) {***

***return numerator / denominator; // This line may throw ArithmeticException***

***}***

***}***

***Error: Division by zero***

***Inside finally block***

***Program execution continues after exception handling***

**Que: What are different ways to handle Multiple Exception in Java?**

**Ans:** In Java, there are several ways to handle multiple exceptions effectively.

**1. Separate Catch Blocks for Each Exception Type:**

* This approach involves using separate catch blocks for each type of exception that may occur.

***try {***

***// Code that may throw exceptions***

***} catch (IOException e) {***

***// Handle IOException***

***} catch (SQLException e) {***

***// Handle SQLException***

***} catch (Exception e) {***

***// Handle other exceptions (catch-all)***

***}***

**2. Multi-Catch (Java 7 and later):**

* Starting from Java 7, you can use the multi-catch feature to catch multiple exception types in a single catch block.
* Multiple exception types are separated by a vertical bar (|).

***try {***

***// Code that may throw exceptions***

***} catch (IOException | SQLException e) {***

***// Handle IOException or SQLException***

***} catch (Exception e) {***

***// Handle other exceptions (catch-all)***

***}***

**3. Nested Try-Catch Blocks:**

* You can use nested try-catch blocks to handle exceptions in a hierarchical manner.
* Inner try-catch blocks can handle specific exceptions, while outer try-catch blocks catch broader exceptions or handle cleanup operations.

***try {***

***try {***

***// Code that may throw exceptions***

***} catch (IOException e) {***

***// Handle IOException***

***}***

***} catch (SQLException e) {***

***// Handle SQLException***

***} catch (Exception e) {***

***// Handle other exceptions (catch-all)***

***}***

**4. Throwing Custom Exceptions:**

* You can create custom exception classes that extend java.lang.Exception to represent specific exceptional conditions in your application.
* By throwing custom exceptions, you can provide more meaningful error messages and handle exceptional scenarios based on your application's requirements.

***public class CustomException extends Exception {***

***public CustomException(String message) {***

***super(message);***

***}***

***}***

***try {***

***// Code that may throw exceptions***

***throw new CustomException("Custom exception message");***

***} catch (CustomException e) {***

***// Handle CustomException***

***} catch (Exception e) {***

***// Handle other exceptions (catch-all)***

***}***

**Que: What is a stack trace and how is it related to an Exception?**

**Ans:** In the context of an exception, when an exception is thrown and not caught at a particular point in the code, Java automatically generates a stack trace for that exception. This stack trace provides valuable information about where the exception occurred and the sequence of method calls leading up to that point.

To print a stack trace in Java, you can use the printStackTrace() method provided by the Throwable class, which is the superclass of all exceptions and errors in Java. Here's an example of how to print a stack trace:

**Que: What is Exception Chaining?**

**Ans:**

Exception Chaining happens when one exception is thrown due to another exception. This helps developers to identify under what situation an Exception was thrown that in turn caused another Exception in the program. For example, we have a method that reads two numbers and then divides them. The method throws ArithmeticException when we divide a number by zero. While retrieving the denominator number from the array, there might have been an IOException that prompted to return of the number as 0 that resulted in ArithmeticException. The original root cause in this scenario was the IOException. The method caller would not know this case and they assume the exception was due to dividing a number by 0. Chained Exceptions are very useful in such cases. This was introduced in JDK 1.4.

**Que: Can we have statements between try, catch and finally blocks?**

**Ans:** No. This is because they form a single unit.

**Que: What happens when an exception is thrown by the main method?**

**Ans:** When there is an exception thrown by the main() method if the exception is not handled, then the program is terminated by the Java Runtime, and the exception message along with the stack trace is printed in the system console.

**Que: Is it possible to throw checked exceptions from a static block?**

**Ans:** We cannot throw a check exception from a static block. However, we can have try-catch logic that handles the exception within the scope of that static block without rethrowing the exception using the throw keyword. The exceptions cannot be propagated from static blocks because static blocks are invoked at the compiled time only once and no method invokes these blocks.

**Que: What happens to the exception object after exception handling is complete?**

**Ans:** The exception object will be garbage collected. When an exception is thrown and not caught, the Java Virtual Machine (JVM) creates an exception object to represent the error condition. This object contains information about the exception, such as the type of exception, error message, stack trace, and other relevant details.

**Que: What does JVM do when an exception occurs in a program?**

**Ans**: When there is an exception inside a method, the method creates and passes (throws) the Exception object to the JVM. This exception object has information regarding the type of exception and the program state when this error happens. JVM is then responsible for finding if there are any suitable exception handlers to handle this exception by going backwards in the call stack until it finds the right handler. If a matching exception handler is not found anywhere in the call stack, then the JVM terminates the program abruptly and displays the stack trace of the exception.

**Que: Does the finally block always get executed in the Java program?**

**Ans:** The finally block in Java is designed to execute code that needs to run regardless of whether an exception occurs or not within a try-catch block. However, there are certain cases where the finally block may not execute, such as:

**1. System.exit() Called**: If the System.exit() method is called in the try or catch block, the program terminates immediately, and the finally block is not executed. This is because System.exit() halts the JVM and does not allow further execution of code.

Example:

***try {***

***// Code that may throw an exception***

***System.exit(0); // Program terminates here, finally block not executed***

***} catch (Exception e) {***

***// Handle the exception***

***} finally {***

***// This block will not be executed if System.exit() is called***

***}***

**2. Infinite Loop or Endless Execution:** If there is an infinite loop or endless execution within the try or catch block (e.g., due to a logical error), the program may not reach the finally block because it remains stuck in the loop indefinitely.

Example:

***try {***

***while (true) {***

***// Code that keeps executing infinitely***

***}***

***} catch (Exception e) {***

***// Handle the exception***

***} finally {***

***// This block may not be executed due to infinite loop***

***}***

**3. Thread Interruption:** If a thread executing the try-catch-finally block is interrupted using Thread.interrupt() or similar methods, it may cause the finally block to be bypassed. However, note that interrupting a thread does not necessarily prevent the finally block from executing, as it depends on how the interruption is handled within the thread.

**Que: Will the Finally block is executed if try and catch block is having return statements in it .**

**Ans:** Yes, the finally block is executed even if a try block contains return statements along with catch blocks in Java. The purpose of the finally block is to ensure that certain code is executed regardless of whether an exception is thrown or not, and regardless of any return statements encountered in the try or catch blocks.

**Que: Why it is always recommended to keep the clean-up activities like closing the I/O resources or DB connections inside a finally block?**

**Ans:** Because of following reason:

**1. Guaranteed Execution:** The finally block is designed to execute code that should always run, regardless of whether an exception occurs or not.

**2. Exception Safety:** By placing cleanup code in the finally block, you ensure that resources are properly released, regardless of whether an exception is thrown during the execution of the try block.

**3. Resource Management**: Resources such as file streams, database connections, network connections, etc., need to be properly managed and released when they are no longer needed.

**4. Consistent Code Flow:** Placing cleanup activities in a finally block helps maintain a consistent code flow and ensures that cleanup operations are performed in a predictable manner.

**Que: Are we allowed to use only try blocks without a catch and finally blocks?**

**Ans:** Yes, in Java, it is possible to use a try block without catch and finally blocks, but there are some important considerations to keep in mind:

**1. try-catch Block:** A try block must be followed by either a catch block or a finally block (or both).

**2. try-finally Block**: If you don't want to handle exceptions within the try block but still need to ensure that certain cleanup operations are performed, you can use a try-finally block.

**Que: What are different scenarios where “Exception in thread main” types of error could occur?**

**Ans**: The "Exception in thread 'main'" error message in Java typically indicates an uncaught exception that occurred in the main thread of the program.

**1. Unchecked Exceptions:** If an unchecked exception (subclass of RuntimeException or Error) is thrown in the main thread and not caught or handled properly, it can lead to an "Exception in thread 'main'" error. Examples of unchecked exceptions include NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException, etc.

**2. Unchecked Error Conditions:** Errors such as OutOfMemoryError or StackOverflowError can occur in the main thread due to resource limitations or stack overflow situations.

**3. Missing Main Method**: If the main method signature is incorrect or missing in the main class of the Java program, the JVM cannot find the entry point to start executing the program. This can lead to an "Exception in thread 'main'" error because the JVM cannot locate the main method to begin program execution.

**4. Incorrect Classpath**: If the classpath is not set correctly or if the required classes or resources are not accessible to the JVM during program execution, it can cause errors in the main thread, resulting in an "Exception in thread 'main'" error.

**Que: What are the rules we should follow when overriding a method throwing an Exception?**

**Ans:** Wahi exception propogation aur exception heirary wala ans bta dena.

**Que: What are the rules we should follow when overriding a method throwing an Exception?**

**Ans:** Yes, it is possible to throw an exception inside a lambda expression's body in Java. Lambda expressions can contain statements, including those that may throw exceptions. However, there are some rules and considerations to keep in mind:

**Que: What is the difference between ClassNotFoundException and NoClassDefFoundError?**

**Ans:** ClassNotFoundException and NoClassDefFoundError are both related to class loading issues in Java, but they represent different scenarios and have distinct meanings:

**1. ClassNotFoundException:**

* This exception is thrown when the Java runtime system tries to load a class by its name using the Class.forName() method or by invoking a method such as Classloader.loadClass(), but the class with the specified name cannot be found in the classpath.
* ClassNotFoundException is a checked exception, which means that the compiler requires you to handle or declare it.
* This exception typically occurs at runtime when the class specified in the code is missing from the classpath, such as when trying to load a JDBC driver class or a user-defined class that is not available during runtime.

**2. NoClassDefFoundError:**

* This error occurs when the Java runtime system tries to load a class and finds the class file but encounters an issue while initializing the class or resolving its dependencies.
* NoClassDefFoundError is an unchecked error, which means that it does not need to be explicitly caught or declared.
* This error typically occurs at runtime when a class that was previously available during compilation is no longer available during execution due to classpath issues, missing dependencies, or incompatible changes in the class structure.

**Que: How are the keywords final, finally and finalize different from each other?**

**Ans:** The keywords final, finally, and finalize are distinct in Java and serve different purposes in the language:

**1. final:**

* final is a keyword used to declare constants, methods, and classes in Java.
* When applied to a variable, it indicates that the variable's value cannot be changed once assigned (i.e., it becomes a constant).
* When applied to a method, it indicates that the method cannot be overridden by subclasses.
* When applied to a class, it indicates that the class cannot be subclassed (i.e., it becomes a final class).

**2. finally**:

* finally is a block used in exception handling to define code that should be executed whether an exception is thrown or not.
* It follows the try-catch block and is optional (i.e., a try block can exist without a finally block, or it can have a finally block without a catch block).
* The finally block is executed regardless of whether an exception occurs or not, making it useful for cleanup operations like closing resources (e.g., file streams, database connections).

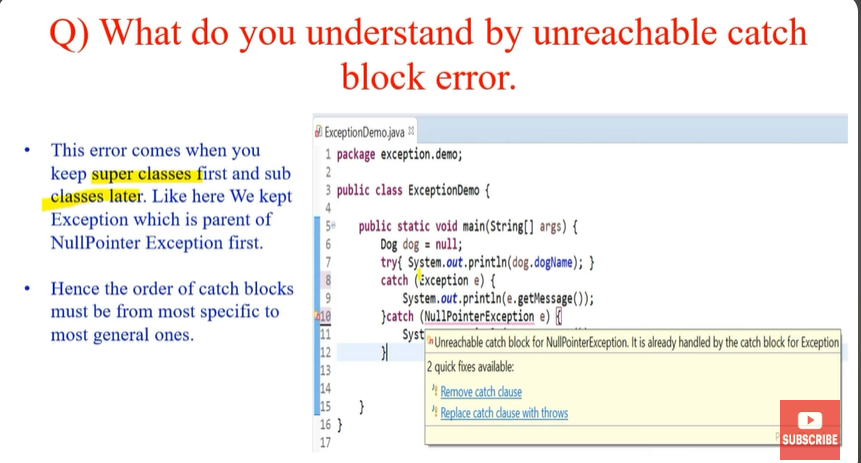
**3. finalize**:

* finalize is a method defined in the Object class that is called by the garbage collector before an object is reclaimed (i.e., before it is destroyed by the garbage collector).
* It is used for performing cleanup or resource releasing operations for an object before it is garbage-collected.
* However, it's important to note that relying on finalize for critical resource cleanup is discouraged because there are no guarantees about when or if the finalize method will be called by the garbage collector.

**Que: What do you understand by unreachable catch block error:**

**Ans:**

This error comes when you keep super classes first and sub classes later.

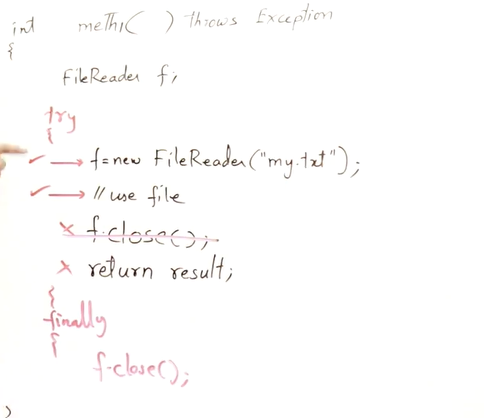


**Que: What is Try with Resource?**

**Ans**: The "try-with-resources" statement is a feature introduced in Java 7 to manage resources automatically and efficiently. It is designed to work with objects that implement the java.lang.AutoCloseable or java.io.Closeable interface, ensuring that these resources are properly closed after being used, even if an exception occurs.

**Try With Resources**

* All the things outside the program is resources: Heap, Printer, Monitor, Network, etc.
* So ethically whenever you require resources you should take it and once the task is done you should return it.
* It the programs responsibility to release the resource.
* Open the file and close it, print from print, and release the printer.
* Java provide in build heap cleaning system called garbage collector.

🡪 Below is the implementation

**How It Works:**

***try (FileReader fileReader = new FileReader(filePath);***

***BufferedReader bufferedReader = new BufferedReader(fileReader)) {***

***String line;***

***while ((line = bufferedReader.readLine()) != null) {***

***System.out.println(line);***

***}***

***} catch (IOException e) {***

***System.err.println("Error reading file: " + e.getMessage());***

***}***

***}***

* When the try block is entered, the resources are initialized in the order they are listed.
* After the try block completes (either normally or due to an exception), the resources are closed in the reverse order of their initialization.
* The close() method of each resource is called automatically, ensuring proper resource management and cleanup.

**Que: How to create custom exception in java ?**

**Ans:** In Java, you can create custom exceptions by extending the Exception class or one of its subclasses (RuntimeException, IOException, etc.) to create your own exception class. Here's a step-by-step guide on how to create a custom exception in Java:

**1. Create a Custom Exception Class:**

Start by creating a new Java class that extends Exception or any appropriate subclass based on your requirements.

***public class CustomException extends Exception {***

***// Constructor with a message***

***public CustomException(String message) {***

***super(message);***

***}***

***// Optional: Additional constructors or methods***

***}***

**2. Throwing the Custom Exception:**

You can throw your custom exception from anywhere in your code when an exceptional condition occurs.

***if (divisor == 0) {***

***throw new CustomException("Division by zero not allowed");***

***}***

***return dividend / divisor;***

***}***

**3. Handling Custom Exceptions:**

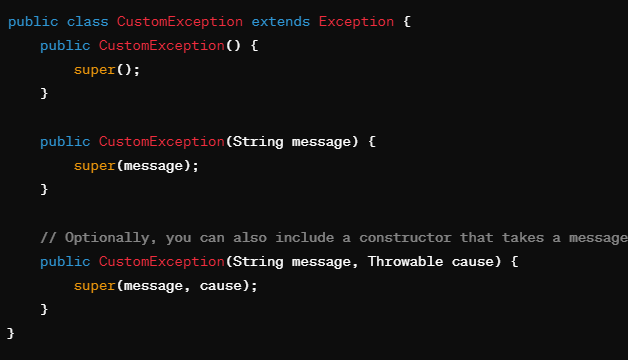
When you throw a custom exception, you can catch it using a try-catch block just like any other exception.

**Que: What are the steps to create custom Exception Handling In java?**

**Ans:** Creating custom exception handling in Java involves creating your own exception class that extends either Exception or one of its subclasses like RuntimeException. Here's a step-by-step guide on how to do this:

**1. Create a Custom Exception Class:**

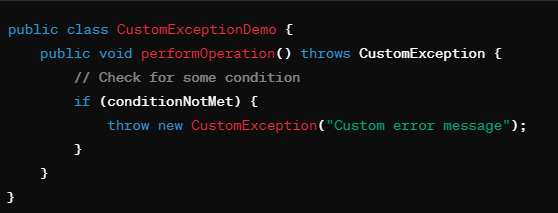
* Decide on a meaningful name for your custom exception class, for example, CustomException.
* Extend Exception or one of its subclasses like RuntimeException.
* Define constructors for your custom exception class, including at least one that takes a message parameter.



* When you create a custom exception class that extends another exception class (like Exception or RuntimeException), using super() in your constructors allows you to call the constructor of the superclass.
* In the context of exception handling, this ensures that the superclass's constructor is properly initialized, which is important for maintaining the exception's integrity and providing useful information about the exception.

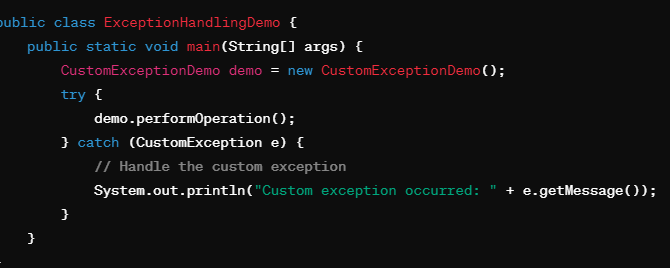
**2. Throwing Custom Exceptions:**

In your code, when a specific condition or error occurs, throw your custom exception.



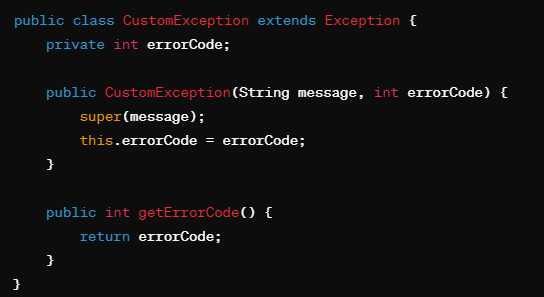
**3. Handling Custom Exceptions:**

When invoking methods that can throw your custom exception, handle it using try-catch blocks.



**4. Optional: Including Custom Information:**

You can include additional information in your custom exception, such as error codes or specific data related to the exception.



**Que: What is Concurrent Modification Exception ?**

**Ans:** A ConcurrentModificationException in Java is a runtime exception that indicates that a collection (such as a List, Set, or Map) has been structurally modified (i.e., elements added, removed, or modified) during a concurrent traversal (e.g., using an iterator) without using the appropriate collection methods.

Cause of ConcurrentModificationException:

The most common cause of ConcurrentModificationException is modifying a collection directly (adding, removing, or modifying elements) while iterating over it using an iterator.

**How to Avoid ConcurrentModificationException:**

* To avoid ConcurrentModificationException, use the appropriate methods provided by the collection classes for adding, removing, or modifying elements while iterating.
* For example, use the Iterator's remove() method instead of the collection's remove() method to safely remove elements during iteration.

**Que: What is Null Pointer Exception and How to handle it ?**

**Ans:** To avoid NullPointerException in Java, you need to follow best practices and defensive programming techniques to ensure that your code handles potential null values appropriately.

**1. Check for Null Before Accessing Objects:**

* Always check if an object reference is null before accessing its methods or fields. This is especially important when dealing with objects returned from methods or obtained from external sources.

**String str = null;**

**if (str != null) {**

**// Safe to access methods or fields of str**

**System.out.println(str.length());**

**}**

**2. Use Optional or Objects.requireNonNull:**

* Java 8 introduced the Optional class, which provides a cleaner way to handle potentially null values.
* Alternatively, you can use Objects.requireNonNull to explicitly check for null values and throw an exception if null is encountered.

**3. Use Try-Catch Blocks for Potential NullPointerExceptions:**

* In cases where you expect a method call to potentially throw NullPointerException, wrap the code in a try-catch block to handle the exception gracefully.

