**What are generics?**

Generics is a programming language feature that allows developers to create flexible, reusable, and type-safe code by using placeholder types in classes, interfaces, or methods. This concept allows the same code to work with multiple different data types without having to explicitly specify the data type during implementation. Generics help reduce code duplication and improve type safety, as they enforce compile-time type checks, ensuring that the objects being used are of the correct type. They are commonly used in collection classes, like lists, sets, and dictionaries, where the same code can be used to handle collections of various data types.

**we change the scope of the overridden method in the subclass for private, public, default Can and protected? Explain how can it be changed for each scope?**

In Java, when you override a method in a subclass, you must follow certain rules related to the access level (scope) of the overridden method. The overriding method in the subclass cannot have a more restrictive access level than the method in the superclass. However, it can have the same or a broader access level. Here's how the scope of the overridden method can be changed in a subclass for each access level:

Private: Private methods are not visible outside their class and cannot be overridden in a subclass. Private methods are only accessible within the class they are defined in.

Public: Public methods are accessible from any class and can be overridden in a subclass with the same (public) access level. You cannot reduce the scope of a public method when overriding it.

Default (package-private): Default access level methods are accessible within the same package. When overriding a default method in a subclass, you can either keep the same default access level or increase the visibility to public. You cannot reduce the scope to private or protected.

Protected: Protected methods are accessible within the same package and by subclasses outside the package. When overriding a protected method in a subclass, you can either keep the same protected access level or increase the visibility to public. You cannot reduce the scope to private or default.

In summary, while overriding a method in a subclass, you can maintain or broaden the access level, but you cannot reduce it. And remember, private methods cannot be overridden.

**What is the covariant return type?**

Covariant return type refers to the ability of a method in a subclass to override a method in its superclass while having a more specific (or derived) return type than the one in the superclass. In Java, covariant return types were introduced in JDK 5.0.With covariant return types, the overriding method in the subclass can return a subtype of the return type declared in the method it's overriding in the superclass. This feature increases the flexibility and readability of the code, making it easier to work with class hierarchies.

**Can we override the static and private methods? Why?**

No, you cannot override static and private methods in Java. Here's why:

Static methods: Static methods are bound to the class rather than an instance of the class. This means that they are resolved at compile-time using static binding, rather than at runtime using dynamic binding, which is used for instance methods. When you define a static method with the same signature in a subclass, you are not overriding the method but hiding it. This is known as method hiding. Since the method resolution is done at compile-time, the method to be called is determined by the reference type, not the actual object type.

Private methods: Private methods are accessible only within the class they are defined in and are not visible to subclasses. Since they are not visible to subclasses, they cannot be overridden. If you define a method with the same signature as a private method in the superclass, it is considered a separate method and not an override.

**Difference between String Buffer and StringBuilder**

Both StringBuffer and StringBuilder are classes in Java used for creating and manipulating strings, particularly when concatenating or modifying strings frequently. They are mutable, unlike the String class, which is immutable. However, there are some key differences between StringBuffer and StringBuilder:

**Synchronization**: The primary difference between StringBuffer and StringBuilder lies in their thread safety. StringBuffer is synchronized, which means that its methods are thread-safe, and only one thread can access them at a time. This ensures the consistency of the buffer's state in a multi-threaded environment. On the other hand, StringBuilder is not synchronized, making it not thread-safe. As a result, multiple threads can access and modify its methods simultaneously, which can cause inconsistencies in a multi-threaded environment.

**Performance**: Due to the lack of synchronization in StringBuilder, it is generally faster than StringBuffer. Since StringBuffer needs to maintain synchronization, it adds some overhead, which can result in slower performance, especially in single-threaded environments where synchronization is not needed. If thread safety is not a concern, StringBuilder is the preferred choice for performance reasons.

**Use cases**: Choose StringBuffer when you need to create and manipulate strings in a multi-threaded environment where thread safety is a requirement. In contrast, use StringBuilder when working in a single-threaded environment or when thread safety is not a concern, as it provides better performance.

In summary, the main differences between StringBuffer and StringBuilder are synchronization and performance. StringBuffer is synchronized and thread-safe but slower, while StringBuilder is not synchronized, making it not thread-safe but faster. Choose the appropriate class based on your application's requirements regarding thread safety and performance.

**Difference between String class and String Buffer?**

String and StringBuffer are both classes in Java that deal with string manipulation. However, they have some fundamental differences in terms of immutability, performance, and methods:

**Immutability:**

String: The String class is immutable, which means that once an instance of the String class is created, its value cannot be changed. When you perform operations like concatenation or substring on a String, a new String object is created, and the original String remains unchanged.

StringBuffer: The StringBuffer class is mutable, which means you can modify its content without creating a new object. This is particularly useful when you need to perform frequent string modifications or concatenations.

**Performance:**

String: Since String objects are immutable, every modification or concatenation operation creates a new object. This can lead to performance issues, particularly when concatenating or modifying strings in a loop.

StringBuffer: As StringBuffer is mutable, it allows for more efficient string manipulation, especially when dealing with large strings or multiple concatenation operations. It allocates extra memory when created, which can be filled up as the string grows, avoiding the creation of multiple objects.

**Methods:**

String: The String class provides various methods for string manipulation, such as concat(), substring(), length(), indexOf(), trim(), etc. However, since it is immutable, methods that seem to modify a String object actually create a new object with the desired modifications.

StringBuffer: The StringBuffer class has methods like append(), insert(), delete(), replace(), and reverse() that allow direct modification of the object's content without creating new objects.

**Synchronization:**

String: As String objects are immutable, there is no need for synchronization, making them inherently thread-safe.

StringBuffer: The StringBuffer class is synchronized, which means that its methods are thread-safe and can be safely used in a multi-threaded environment.

In summary, the main differences between the String and StringBuffer classes are immutability, performance, and methods. Use the String class when dealing with constant or small strings, and use the StringBuffer class when you need efficient string manipulation, particularly when concatenating or modifying strings frequently.

**Can we try without a catch block in Java?**

Yes, you can have a try block without a catch block in Java, but you must include a finally block. The finally block is used to execute a set of statements regardless of whether an exception occurs or not**.**

**What is try with the resource?**

try-with-resources is a feature introduced in Java 7 that allows for automatic resource management. It is a convenient way to ensure that resources, such as files, sockets, or database connections, are properly closed after they are used. The try-with-resources statement ensures that each resource declared within the parentheses of the try block is closed when the block exits, whether it exits normally or due to an exception.

**Can we modify the throws clause of the superclass method while overriding it in the subclass?**

Yes, you can modify the throws clause of a superclass method while overriding it in a subclass, but there are certain rules that you need to follow to maintain proper method overriding:If the superclass method does not declare any checked exceptions, the overriding method in the subclass can only declare unchecked exceptions or not declare any exceptions at all.If the superclass method declares checked exceptions, the overriding method in the subclass can declare the same checked exceptions, a subset of the checked exceptions declared by the superclass method, or no exceptions. The overriding method cannot declare new checked exceptions that are not declared by the superclass method.

The overriding method can declare any unchecked exceptions (i.e., exceptions that are subclasses of java.lang.RuntimeException), irrespective of the exceptions declared by the superclass method**.**

**Example:**

class SuperClass {

void exampleMethod() throws IOException {

// ...

}

}

class SubClass extends SuperClass {

// This is a valid overriding

@Override

void exampleMethod() throws FileNotFoundException {

// ...

}

}

**What is an association, aggregation, and composition in UML?**

Association:

Association represents a simple relationship between classes or objects where one class is associated with another class through a link. It can be bi-directional or uni-directional and can have various multiplicity values (one-to-one, one-to-many, many-to-one, or many-to-many).

Association can represent relationships like "uses," "has," or "knows" between classes or objects.

Example: A Student class is associated with a Course class; a student can be enrolled in multiple courses, and a course can have multiple students.

Aggregation:

Aggregation is a special type of association that represents a "whole-part" or "has-a" relationship between classes or objects, where one class is a container, and the other class is its part or component.

Aggregation is a weaker form of relationship, which implies that the container object can exist independently of the component objects. When the container object is destroyed, the component objects are not necessarily destroyed.

Example: A Department class is an aggregate of Employee objects; a department can have multiple employees, but an employee can still exist even if a department is dissolved.

Composition:

Composition is also a "whole-part" relationship like aggregation, but it is a stronger form of relationship. It represents a stricter "has-a" relationship between classes or objects, where the lifetime of the component objects is tightly coupled to the lifetime of the container object.

In composition, when the container object is destroyed, the component objects are also destroyed, as they cannot exist independently of the container object.

Example: A Car class is a composition of an Engine class; a car has an engine, and the engine cannot exist independently of the car. When a car is destroyed, the engine is also destroyed.

In summary, association represents a general relationship between classes or objects, aggregation represents a "whole-part" relationship with independent lifetimes, and composition represents a "whole-part" relationship with dependent lifetimes**.**

**Difference between final, finally and finalize()?**

**final:**

final is a keyword that can be used with classes, methods, and variables.

When used with a class, final makes the class non-extendable, meaning that no other class can inherit from it.

When used with a method, final prevents the method from being overridden in a subclass.

When used with a variable, final makes the variable a constant, meaning that its value cannot be changed after initialization.

**finally:**

finally is a block associated with a try-catch statement in exception handling.

The code inside the finally block is guaranteed to execute, regardless of whether an exception was thrown or caught within the try or catch blocks.

finally is often used for cleaning up resources or completing tasks that must be done, irrespective of whether an exception occurred or not.

**finalize():**

finalize() is a method in the java.lang.Object class, which is the superclass of all classes in Java. It is called by the garbage collector before an object is destroyed and its memory is reclaimed.

The finalize() method can be overridden in a class to provide custom cleanup logic, such as releasing resources or closing connections.

However, it's important to note that relying on finalize() for cleanup is not recommended, as there is no guarantee when the garbage collector will run or if the finalize() method will be called at all. Instead, it's better to use explicit resource management techniques like the try-with-resources statement.

In summary, final is a keyword for creating non-extendable classes, non-overridable methods, and constant variables; finally is a block in exception handling that always executes; and finalize() is a method called by the garbage collector before an object is destroyed, which can be overridden for custom cleanup tasks.

**Difference between Vector and ArrayList?**

Vector and ArrayList are both classes in Java that implement the List interface, which is part of the Java Collections Framework. They can be used to store and manage lists of objects. However, there are some key differences between them:

Synchronization:Vector is synchronized, which means that its methods are thread-safe and can be safely accessed by multiple threads without causing any data inconsistency. On the other hand, ArrayList is no synchronized, and concurrent access by multiple threads may lead to data inconsistency. If you require a thread-safe list, you can use Vector, or you can use Collections.synchronizedList() to create a synchronized version of an ArrayList.

Performance:

Due to the synchronization in Vector, its operations may have a performance overhead when compared to ArrayList. Since ArrayList is not synchronized, its operations are generally faster. If you don't require thread-safety, it is recommended to use ArrayList for better performance.

Capacity Increment:

When a Vector or an ArrayList needs to grow in size, they both increase their capacity. However, they handle this capacity increment differently. Vector has a default capacity increment of 100% (doubles its size), or you can specify a custom increment during its creation. In contrast, ArrayList increases its capacity by 50% of its current size (1.5 times its size) by default. This behavior can affect the performance and memory usage of the collections.

Legacy:

Vector is a legacy class, introduced in Java 1.0, while ArrayList was introduced in Java 1.2 as a part of the Java Collections Framework. It's generally recommended to use ArrayList over Vector when you don't need synchronization, as it is more modern and offers better performance.

In summary, the main differences between Vector and ArrayList are synchronization, performance, capacity increment, and legacy status. Use ArrayList for most use cases, but consider Vector or a synchronized ArrayList if you require thread-safety.

**What are the different ways to make ArrayList methods synchronized?**

There are different ways to make the methods of ArrayList synchronized in Java, which can be useful in cases where you need to ensure thread-safety. Here are three ways to do it:

Use Collections.synchronizedList():

The Collections class provides a static method called synchronizedList() that returns a synchronized wrapper of an ArrayList. This wrapper list is thread-safe, and all the operations on it are synchronized.

Note that all the methods of the synchronized list returned by Collections.synchronizedList() are synchronized, so they have a performance overhead compared to an unsynchronized ArrayList.

Use Vector:Vector is another class in Java that implements the List interface and provides thread-safety through synchronization. You can use Vector instead of ArrayList if you need thread-safety.All thmethods of Vector are synchronized by default, so you don't need to do anything extra to make them thread-safe.

Synchronize individual methods:

If you only need to synchronize some specific methods of an ArrayList, you can use the synchronized keyword to synchronize them manually. For example, you can synchronize the add() method of an ArrayList like this:

This makes the add() method thread-safe, but it's important to note that other methods of the ArrayList are still not synchronized and can cause data inconsistency if accessed by multiple threads simultaneously. If you need to synchronize multiple methods, it's easier and safer to use one of the previous two methods to make the entire list synchronized.

**In Java 8, explain how Hasp Map internally works?**

In Java 8, HashMap internally works using an array of Node objects to store the key-value pairs, where each Node object represents a single entry in the HashMap. Each Node object contains a key, a value, and a reference to the next Node object in the same bucket.

HashMap uses a hashing function to map each key to a specific index in the array, which is called a bucket. The hashing function is designed to distribute the keys uniformly across the array, which reduces the number of collisions (cases where multiple keys map to the same bucket) and improves the performance of the HashMap.

To retrieve the value associated with a key, HashMap first calculates the bucket index using the hashing function, and then iterates over the Node objects in the bucket until it finds the Node object that matches the key. If there are multiple Node objects in the bucket, HashMap uses the equals() method to compare the keys until it finds a match.

In Java 8, HashMap uses a technique called treeification to optimize the performance of the get() and put() operations. When a bucket contains too many Node objects (by default, more than 8), HashMap converts the linked list of Node objects into a balanced binary tree using the red-black tree algorithm. This reduces the time complexity of the get() and put() operations from O(n) to O(log n), where n is the number of entries in the bucket.

HashMap also provides methods like putIfAbsent(), compute(), merge(), and forEach() that make use of the new lambda expressions and streams API introduced in Java 8. These methods allow for more concise and expressive code, and can help improve the readability and maintainability of the code that uses HashMap.

**Difference between Hash table and Hash Map?**

HashTable and HashMap are both classes in Java that implement the Map interface, which is part of the Java Collections Framework. They are used to store and manage key-value pairs, where each key is associated with a value. However, there are some key differences between them:

Synchronization:

HashTable is synchronized, which means that its methods are thread-safe and can be safely accessed by multiple threads without causing any data inconsistency. On the other hand, HashMap is not synchronized, and concurrent access by multiple threads may lead to data inconsistency. If you require a thread-safe map, you can use HashTable, or you can use Collections.synchronizedMap() to create a synchronized version of a HashMap.

Performance:

Due to the synchronization in HashTable, its operations may have a performance overhead when compared to HashMap. Since HashMap is not synchronized, its operations are generally faster. If you don't require thread-safety, it is recommended to use HashMap for better performance.

Null keys and values:

HashTable does not allow null keys or values, and will throw a NullPointerException if you try to put or get a null key or value. In contrast, HashMap allows one null key and any number of null values.

Enumeration:

HashTable provides an Enumeration that can be used to iterate over its keys and values, while HashMap does not provide an Enumeration. Instead, HashMap provides an enhanced for loop and an iterator that can be used to iterate over its keys, values, or entries.

In summary, the main differences between HashTable and HashMap are synchronization, performance, null keys and values, and enumeration. Use HashMap for most use cases, but consider HashTable or a synchronized HashMap if you require thread-safety. However, keep in mind that the use of HashTable is discouraged in favor of ConcurrentHashMap for concurrent access due to its performance limitations.

**Difference between fail fast and fail-safe iterator?**

Fail-fast and fail-safe are two approaches to handle concurrent modification in iterators in Java. Here are the key differences between them:

Concurrent Modification:When an iterator is created, it holds a reference to the underlying collection. If the collection is modified during iteration, the iterator may become invalid and throw a ConcurrentModificationException. This can happen when elements are added, removed or modified during the iteration.

Behavior:

A fail-fast iterator detects concurrent modifications at the time they occur and throws a ConcurrentModificationException immediately. This approach is preferred in cases where you need to detect and handle concurrent modifications quickly to avoid data inconsistency.

A fail-safe iterator does not throw a ConcurrentModificationException even if the underlying collection is modified during the iteration. Instead, it creates a copy of the collection and iterates over the copy, ensuring that the original collection remains unmodified. This approach is preferred in cases where you need to avoid throwing exceptions and continue the iteration despite concurrent modifications.

Performance:

A fail-fast iterator is generally more efficient than a fail-safe iterator, as it does not need to create a copy of the collection. However, a fail-safe iterator may be more appropriate in situations where concurrent modifications are expected to be frequent, or if the collection is large and copying it would have a significant performance impact.

Supported Collections:

Fail-fast iterators are supported by most collection classes in Java, such as ArrayList, LinkedList, HashSet, and TreeSet. Fail-safe iterators are supported by some concurrent collection classes, such as ConcurrentHashMap and CopyOnWriteArrayList.

In summary, the main differences between fail-fast and fail-safe iterators are in how they handle concurrent modification and their behavior during iteration. Fail-fast iterators throw a ConcurrentModificationException immediately, while fail-safe iterators create a copy of the collection and iterate over it. The choice between the two approaches depends on the specific requirements of the application, such as the need for fast detection of concurrent modifications or the ability to continue the iteration despite concurrent modifications.

**Can we start the thread twice?**

No, we cannot start a thread twice in Java. If we try to start a thread more than once, it will throw an IllegalThreadStateException. Once a thread has been started and executed, it cannot be started again.

If you need to execute the same code multiple times, you can create a new thread object each time and start it separately. Alternatively, you can reuse the same thread object but make sure to wait for the thread to complete its previous execution before starting it again.

It's important to note that creating and starting a new thread can have a performance overhead, so it's generally recommended to reuse the same thread object if possible, rather than creating a new one each time. However, you should ensure that the thread has completed its previous execution before starting it again, otherwise it can lead to unpredictable behavior and data inconsistency.

**What are the different ways to create a thread in java? Which one is preferred?**

Extending the Thread class:

You can create a new class that extends the Thread class and override the run() method to define the code that the thread will execute. Then, you can create an instance of this class and call its start() method to start the thread

Implementing the Runnable interface:

You can create a new class that implements the Runnable interface and override its run() method to define the code that the thread will execute. Then, you can create an instance of the Thread class and pass your Runnable object to its constructor. Finally, you can call the start() method of the Thread object to start the thread.

**What are the different states a thread will go through?**

A thread in Java can go through several different states during its lifecycle. Here are the different states that a thread can go through:

New: When a thread is created but not yet started, it is in the new state. In this state, the thread has been allocated memory and its initial values have been set, but it has not yet started executing.

Runnable: When a thread is started, it moves into the runnable state. In this state, the thread is eligible to run, but it may not necessarily be executing at any given moment. When the thread gets its turn to run, it moves into the running state.

Running: When a thread is executing its code, it is in the running state. In this state, the thread is actively executing its instructions.

Blocked: Sometimes a thread may be unable to proceed with its execution, such as when it is waiting for I/O or for another thread to release a lock. In these cases, the thread moves into the blocked state. While a thread is blocked, it cannot make progress until the condition that caused the block is resolved.

Waiting: A thread may also move into a waiting state when it is waiting for a specific condition to be met. In this state, the thread is not actively doing anything, but is waiting for a specific event or signal to occur.

Timed Waiting: Similar to the waiting state, a thread may move into a timed waiting state when it is waiting for a specific condition to be met, but with a time limit. In this state, the thread will wait for the specified time period before resuming its execution.

Terminated: When a thread completes its execution or is terminated prematurely, it moves into the terminated state. In this state, the thread is no longer running and its resources have been released.

Understanding these different states is important when writing multithreaded applications, as it helps you to understand how threads interact with each other and how to manage their execution effectively.

**What is Serialization? How do we achieve it?**

Serialization is the process of converting an object into a stream of bytes, which can be stored on disk, sent over a network, or otherwise persisted or transmitted. Deserialization is the reverse process, where the bytes are converted back into an object.

By implementing the Serializable interface, we are indicating that instances of this class can be serialized. The Java serialization mechanism will take care of the rest, converting the object into a stream of bytes that can be saved to disk, sent over the network, or otherwise persisted or transmitted.

**What is immutable class? Is String class immutable**?

An immutable class is a class whose instances cannot be modified after they are created. Once an instance of an immutable class is created, its state cannot be changed.

In Java, an immutable class typically has the following properties:

All instance variables are marked as final so that they cannot be changed after object creation.

The class does not provide any public methods that can modify its state.

If the class provides methods that modify its state, they make a copy of the object's state and return a new instance with the modified state, rather than modifying the existing instance.

The String class in Java is immutable. Once a String object is created, its value cannot be changed

**Do immutable classes thread safe? If yes then how?**

Immutable classes are inherently thread-safe because their state cannot be modified after they are created. This means that multiple threads can access an immutable object at the same time without any synchronization, and there will be no race conditions or other concurrency issues.

Since immutable objects cannot be modified, there is no risk of multiple threads accessing the same object and trying to modify it simultaneously. This eliminates the need for synchronization and makes immutable classes ideal for use in multi-threaded environments.

For example, consider the following immutable class:

java

Copy code

public final class ImmutableClass {

private final int id;

private final String name;

public ImmutableClass(int id, String name) {

this.id = id;

this.name = name;

}

public int getId() {

return id;

}

public String getName() {

return name;

}

}

In this example, the ImmutableClass is immutable because its state cannot be modified after it is created. The class has two private instance variables, id and name, both of which are marked as final and cannot be changed after the object is created. The class provides getter methods for both instance variables, but no setter methods.

Because the ImmutableClass is immutable, it is inherently thread-safe. Multiple threads can access an instance of the ImmutableClass at the same time without any need for synchronization or locking, and there will be no race conditions or other concurrency issues.

In summary, immutable classes are thread-safe because their state cannot be modified after they are created. This eliminates the risk of race conditions and other concurrency issues, and makes them ideal for use in multi-threaded environments.

**Can we call the garbage collector explicitly? Will it trigger the garbage collector?**

Yes, we can call the garbage collector explicitly in Java using the System.gc() method. This method requests that the JVM run the garbage collector to free up memory that is no longer being used.

However, calling the System.gc() method does not guarantee that the garbage collector will run immediately or even at all. The JVM may choose to delay or skip garbage collection if it determines that there is enough memory available or if it is not an optimal time to perform garbage collection.

Furthermore, calling System.gc() too frequently or unnecessarily can negatively impact performance, as the garbage collector can be a resource-intensive process that can cause noticeable pauses in application execution.

In general, it is best to let the JVM manage memory and garbage collection automatically, without explicitly calling the System.gc() method. Modern JVMs are highly optimized for memory management and garbage collection, and are capable of efficiently freeing up memory as needed.

However, in some cases, such as when dealing with large objects or performing intensive memory operations, it may be useful to explicitly call System.gc() to help manage memory usage and prevent out-of-memory errors.

**Example:**

public class GarbageCollectionExample {

public static void main(String[] args) {

// Create some objects

Object obj1 = new Object();

Object obj2 = new Object();

Object obj3 = new Object();

// Set the objects to null to release their references

obj1 = null;

obj2 = null;

obj3 = null;

// Call the garbage collector explicitly

System.gc();

}

}

**What are Java 8 features? Explain all of them with examples?**

Java 8 introduced many new features and enhancements, including:

Lambda Expressions

Functional Interfaces

Streams API

Default Methods

Method References

Date and Time API

Optional Class

Nashorn JavaScript Engine

Let's go through each of these features in more detail:

Lambda Expressions:

Lambda expressions allow us to write more concise and functional code by replacing anonymous inner classes with a more compact syntax. Lambda expressions are essentially a way to pass code as data, allowing us to write functions that can be passed as arguments to other functions.

java

Copy code

// Example of a lambda expression that squares an integer

Function<Integer, Integer> square = x -> x \* x;

int result = square.apply(5); // result = 25

Functional Interfaces:

Functional interfaces are interfaces that contain exactly one abstract method. They are used extensively in Java 8 to represent lambdas and other functional programming constructs. Java 8 defines several built-in functional interfaces, such as Predicate, Function, and Supplier.

java

Copy code

// Example of a functional interface that squares an integer

@FunctionalInterface

public interface IntSquare {

int square(int x);

}

IntSquare square = x -> x \* x;

int result = square.square(5); // result = 25

Streams API:

The Streams API is a new API for processing collections in a functional style. Streams allow us to express complex data transformations and aggregations using a fluent and declarative syntax.

java

Copy code

// Example of using Streams to filter and collect elements

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

List<String> filteredNames = names.stream()

.filter(name -> name.startsWith("A"))

.collect(Collectors.toList()); // filteredNames = ["Alice"]

Default Methods:

Default methods are methods that are declared in an interface with a default implementation. They allow us to add new methods to an interface without breaking existing implementations.

java

Copy code

// Example of a default method in an interface

public interface Animal {

void speak();

default void move() {

System.out.println("The animal is moving");

}

}

public class Cat implements Animal {

public void speak() {

System.out.println("Meow!");

}

}

Cat cat = new Cat();

cat.speak(); // prints "Meow!"

cat.move(); // prints "The animal is moving"

Method References:

Method references allow us to pass a method as a reference, without calling it. They can be used to create more concise and readable code when working with lambdas.

java

Copy code

// Example of a method reference

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

names.stream()

.map(String::toUpperCase)

.forEach(System.out::println);

Date and Time API:

The Date and Time API is a new API for working with dates and times in a more convenient and flexible way than the old java.util.Date and java.util.Calendar classes.

java

Copy code

// Example of using the Date and Time API to parse a date

String dateString = "2019-04-01";

LocalDate date = LocalDate.parse(dateString);

System.out.println(date.getDayOfWeek()); // prints "MONDAY"

Optional Class:

The Optional class is a new class that represents a value that may or may not be present. It provides a cleaner and safer way of dealing with null values.

java

Copy code

// Example of using the Optional class to handle null values

String name = null;

Optional<String> optionalName = Optional.ofNullable(name);

System.out.println(optionalName.orElse)

**How to make a pure singleton?**

To create a pure singleton in Java, you need to ensure that only one instance of the class can be created and that this instance is globally accessible.

Explained in the code

**How to make a singleton synchronized?**

To make a singleton synchronized in Java, you need to ensure that only one instance of the class can be created and that this instance is globally accessible, while also ensuring thread safety.

One common approach to creating a synchronized singleton in Java is to use a private constructor, a private static field to hold the single instance of the class, a public static synchronized method to access the instance, and the volatile keyword to ensure that changes to the instance variable are immediately visible to all threads. Here's an