1. What is the difference between JDK, JRE, JVM, and JIT Compiler?

Component	Description
JDK (Java Development Kit)	It contains the tools needed to develop Java applications, including the compiler (javac), debugger, documentation generator, and JRE.
JRE (Java Runtime Environment)	It provides the libraries, Java Virtual Machine (JVM), and other components to run Java applications. It does **not include development tools**.
JVM (Java Virtual Machine)	It is an abstract machine that provides a runtime environment for executing Java bytecode. It converts bytecode to machine code during execution.
JIT (Just-In-Time) Compiler	It is part of the JVM. It improves performance by compiling bytecode into native machine code at runtime.

---

2. What is Java Virtual Machine (JVM)?

- JVM is an **abstract machine** that provides the environment to execute Java applications.
- It converts Java bytecode (produced by the compiler) into machine code, which the operating system can understand.
- JVM ensures Java's **platform independence** by providing a uniform execution environment across different platforms.

---

3. What are the different types of memory areas allocated by JVM?

JVM allocates the following memory areas:

- **Method Area** – Stores class-level information like method code, constants, and static variables.
- **Heap** – Stores objects and instance variables.
- **Stack** – Stores method calls and local variables. Each thread has its own stack.
- **PC (Program Counter) Register** – Keeps track of the current instruction being executed by each thread.
- **Native Method Stack** – Used for native methods written in languages like C/C++.

---

4. What is JIT Compiler?

- JIT compiler is a part of the JVM that improves performance by **converting bytecode into native machine code** at runtime.
- Instead of interpreting bytecode line by line, JIT compiles the entire code and caches it for future use, improving execution speed.

---

5. How is the Java platform different from other platforms?

- Java is a **platform-independent** language because its compiled code (bytecode) can run on any operating system that has a JVM.
- Other platforms (like Windows, Linux) are **hardware-dependent** and need separate compilation for different systems.

---

6. What is platform independence in Java?

- Java code is compiled into **bytecode** (.class file) using the javac compiler.
- Bytecode is not machine-specific; it can run on any system with a compatible JVM, making Java platform-independent.

---

7. How many class loaders are there in Java?

Java has **three class loaders**:

1. **Bootstrap ClassLoader** – Loads core Java classes (java.lang package).
2. **Extension ClassLoader** – Loads classes from the lib/ext directory.
3. **Application ClassLoader** – Loads classes from the application's classpath (classpath).

---

8. What is Delegation Hierarchy Algorithm?

- The delegation hierarchy algorithm ensures that a class loader first delegates the class loading request to its **parent class loader**.
- If the parent cannot load the class, the current class loader attempts to load it.
- The order of delegation is: **Bootstrap → Extension → Application**.

---

9. Can we write the main method as public void static instead of public static void?

- ❌ **No**.
- The correct syntax is:
  public static void main(String[] args)
- The order matters because the static keyword must come before the return type.

---

10. In Java, if we do not specify any value for local variables, what will be the default value of the local variables?

- ❌ **No default value** for local variables.
- If you try to access a local variable without initializing it, **a compile-time error** will occur.
- Example:

```
public class Test {
    public static void main(String[] args) {
        int x;
        System.out.println(x); // Compile-time error: x might not have been initialized
    }
}
```

11. Let's say we run a Java class without passing any arguments. What will be the value of the String array of arguments in the main method?

- If no arguments are passed, the String[] args will be an **empty array** (args.length = 0), but it will **not be null**.
  Example:

```
public class Test {
    public static void main(String[] args) {
        System.out.println(args.length); // Output: 0
    }
}
```

---

12. What is the difference between byte and char data types in Java?

Feature	byte	char
Size	8 bits (1 byte)	16 bits (2 bytes)
Range	-128 to 127	0 to 65,535 (Unicode characters)
Purpose	Stores numeric values (integer)	Stores a single character
Example	byte b = 100;	char c = 'A';

---

13. Ascending order of numeric data types?

Ascending order based on size:

byte (1 byte) < short (2 bytes) < int (4 bytes) < long (8 bytes) < float (4 bytes) < double (8 bytes)

---

14. Can I take multiple classes in a single Java file?

✅ **Yes**, but with some rules:

- You can define multiple classes in a single .java file.
- Only **one class** can be public, and the file name must match the public class name.

Example:

```
public class A {
    public static void main(String[] args) {
        System.out.println("Class A");
    }
}

class B {
    void display() {
        System.out.println("Class B");
    }
}
```

---

15. What is OOPS?

- OOPS stands for **Object-Oriented Programming System**.
- It is a programming model based on objects and classes that allows for better organization and reuse of code.

---

16. What are the main principles of Object-Oriented Programming?

The four main OOPS principles are:

1. **Encapsulation** – Wrapping data and methods together into a single unit (class).
2. **Abstraction** – Hiding implementation details and exposing only the necessary parts.
3. **Inheritance** – Acquiring properties and behaviors from a parent class.
4. **Polymorphism** – Ability to define multiple methods with the same name but different implementations (method overloading and method overriding).

---

17. What is the difference between Object-Oriented Programming Language and Object-Based Programming Language?

Feature	Object-Oriented Programming	Object-Based Programming
Supports Inheritance	✅ Yes	❌ No
Example	Java, C++, Python	JavaScript, VBScript
Encapsulation, Abstraction, Polymorphism	✅ Supported	✅ Supported
Based on Classes	✅ Yes	❌ No (direct object creation)

---

18. In Java, what is the default value of an object reference defined as an instance variable in an object?

- The default value of an object reference is **null** if not initialized.

Example:

```
class Test {
    Object obj;

    void display() {
        System.out.println(obj); // Output: null
    }
}
```

---

19. Why do we need a constructor in Java?

- A constructor is used to **initialize an object** when it is created.
- It assigns default or user-defined values to the object's attributes.
- A constructor:
  - Has the **same name** as the class.
  - **No return type** (not even void).

Example:

```
class Test {
    int x;

    Test() {
        x = 5;
    }

    void display() {
        System.out.println(x); // Output: 5
    }
}
```

---

20. Why do we need a default constructor in Java classes?

- If no constructor is defined, Java provides a **default constructor** automatically.
- If a parameterized constructor is defined, Java **does not create a default constructor** unless you define it explicitly.

Example:

```
class Test {
    Test() {
        System.out.println("Default constructor called");
    }

    public static void main(String[] args) {
        Test obj = new Test(); // Output: Default constructor called
    }
}
```

21. What is the value returned by a Constructor in Java?

- A constructor does **not return any value** (not even void).
- However, it **returns the current instance** of the class implicitly.

Example:

```
class Test {
    Test() {
        System.out.println("Constructor called");
    }
```

```
}

Test obj = new Test(); // Implicitly returns the current object
```

---

22. Can we inherit a Constructor?

❌ **No**

- A constructor is **not inherited** by a subclass.
- However, a subclass can call the parent class constructor using the super() keyword.

Example:

```
class Parent {
    Parent() {
        System.out.println("Parent constructor");
    }
}

class Child extends Parent {
    Child() {
        super(); // Calls parent constructor
        System.out.println("Child constructor");
    }
}
```

---

23. Why constructors cannot be final, static, or abstract in Java?

- **final** – Constructors cannot be overridden, so marking them final is meaningless.
- **static** – Constructors are related to object creation, but static methods belong to the class, not the object.
- **abstract** – Abstract methods have no implementation, but constructors must have a body to initialize an object.

---

24. What is the purpose of 'this' keyword in Java?

- this refers to the **current instance** of the class.
- Used to:
  - Refer to the current class instance variables.
  - Invoke current class methods or constructors.
  - Pass as an argument to a method or constructor.

Example:

```
class Test {
    int x;

    Test(int x) {
        this.x = x; // Refers to the instance variable
    }
}
```

---

25. Explain the concept of a static variable. How is it different from an instance variable?

Feature	Static Variable	Instance Variable
Memory	Stored in **method area**	Stored in **heap**
Belongs To	Class	Object

Feature	Static Variable	Instance Variable
Access	Accessed using class name	Accessed using object reference
Lifetime	Exists until the program ends	Exists until the object is destroyed

---

26. Explain the concept of Inheritance?

- Inheritance allows one class (**subclass**) to acquire the properties and methods of another class (**superclass**).
- Promotes **code reuse** and **polymorphism**.

Example:

```
class Parent {
   void display() {
      System.out.println("Parent method");
   }
}

class Child extends Parent {
   void show() {
      System.out.println("Child method");
   }
}
```

---

27. Which class in Java is superclass of every other class?

✅ Object class

---

28. Why Java does not support multiple inheritance?

- To **avoid ambiguity** and **complexity** caused by the diamond problem.
- Java allows multiple inheritance through **interfaces**.

---

29. In OOPS, what is meant by composition?

- **Composition** is a "Has-A" relationship where one class contains an object of another class.
- Strong association – If the container object is destroyed, the composed object is also destroyed.

Example:

```
class Engine {}
class Car {
   private final Engine engine = new Engine(); // Composition
}
```

---

30. How aggregation and composition are different concepts?

Feature	Aggregation	Composition
Dependency	Weak relationship	Strong relationship
Lifetime	Child object can exist without parent	Child object cannot exist without parent
Example	Student and Department	Car and Engine

---

31. Why there are no pointers in Java?

- To **avoid memory leaks** and **security issues**.
- JVM handles memory management through **garbage collection**.

---

32. What is encapsulation in Java, and why is it considered a fundamental principle of object-oriented programming?

- **Encapsulation** – Wrapping data (variables) and code (methods) into a single unit (class).
- Provides **data hiding** and **control over data access** using **getters and setters**.

Example:

```
class Test {
    private int value;

    public void setValue(int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }
}
```

---

33. What is the purpose of 'super' keyword in Java?

- super refers to the **parent class**.
- Used to:
  - Call parent class constructor.
  - Access parent class methods or variables.

---

34. Is it possible to use this() and super() both in the same constructor?

❌ **No** – this() or super() must be the first statement in a constructor.

---

35. What is the meaning of object cloning in Java?

- clone() method creates a **copy** of an object.
- Implement Cloneable interface and override clone() method.

Example:

```
class Test implements Cloneable {
    int value;

    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

---

36. In Java, why do we use static variables?

- To share a common variable among all objects of a class.

---

37. Why is it not a good practice to create static variables in Java?

- Can cause **memory leaks** if not handled properly.
- Difficult to manage in multithreaded environments.

---

38. What is the purpose of static method in Java?

- Used for **utility methods** that don't depend on instance variables.
- Can be called without creating an object.

---

39. Why do we mark the main method as static in Java?

- JVM calls the main method without creating an object.

---

40. In what scenario do we use a static block?

- Used to **initialize static variables**.
- Executes **before the main method**.

Example:

```
class Test {
   static {
      System.out.println("Static block");
   }
}
```

---

41. Is it possible to execute a program without defining a main() method?

✅ Yes, using **static blocks** (before Java 7).

❌ No, after Java 7+.

---

42. What happens when static modifier is not mentioned in the signature of the main method?

❌ **Compile-time error** – main method must be public static void.

---

43. What is the difference between static method and instance method in Java?

Feature	Static Method	Instance Method
Access	Called using class name	Called using object reference
Data	Works on static data	Works on instance data
Inheritance	Cannot be overridden	Can be overridden

---

44. What is the other name of Method Overloading?

✅ **Compile-time polymorphism**

---

45. How will you implement method overloading in Java?

- Same method name with **different parameters**.

---

46. What kinds of argument variations are allowed in Method Overloading?

- Type, number, and order of parameters.

---

47. Why it is not possible to do method overloading by changing return type?

- **Ambiguity** during compilation.

---

48. Is it allowed to overload main() method in Java?

✅ Yes – But JVM calls only the standard main(String[] args).

---

49. How do we implement method overriding in Java?

- Define a method in a subclass with the same signature as in the parent class.

---

50. Are we allowed to override a static method in Java?

❌ **No** – Static methods are not part of the object, so they cannot be overridden.

51. Why Java does not allow overriding a static method?

- Static methods belong to the **class** (not an instance).
- Overriding is based on the runtime object type, but static methods are resolved at **compile time** (static binding).

---

52. Is it allowed to override an overloaded method?

✅ **Yes** – Overloading and overriding are independent concepts.

- Overloading – Same method name with different parameters in the same class.
- Overriding – Same method signature in a subclass.

---

53. What is the difference between method overloading and method overriding in Java?

Feature	Method Overloading	Method Overriding
Parameters	Must be different	Must be the same
Return Type	Can be different	Must be the same (or covariant)
Inheritance	Not required	Requires inheritance
Binding	Compile-time (Static)	Runtime (Dynamic)
Access Modifier	Can be different	Must be same or more visible

---

54. Does Java allow virtual functions?

✅ **Yes** –

- By default, all **non-static methods** in Java are virtual functions.
- They are dynamically bound at runtime, except for final, static, and private methods.

---

55. What is meant by covariant return type in Java?

- When overriding a method, the return type in the subclass can be a **subtype** of the parent class return type.

Example:

```
class Parent {
    Parent get() {
        return this;
    }
}

class Child extends Parent {
    @Override
    Child get() {
        return this;
    }
}
```

- **Polymorphism**

56. What is Runtime Polymorphism?

- Also known as **Dynamic Method Dispatch**.
- Achieved through **method overriding**.
- The method to execute is determined at **runtime**.

Example:

```
class Parent {
    void display() {
        System.out.println("Parent");
```

```
    }
}

class Child extends Parent {
    @Override
    void display() {
        System.out.println("Child");
    }
}

Parent obj = new Child();
obj.display(); // Output: Child
```

---

57. Is it possible to achieve Runtime Polymorphism by data members in Java?

❌ **No**

- Data members are resolved at **compile-time** (static binding).

---

58. Explain the difference between static and dynamic binding?

Binding Type	Time of Resolution	Example
Static Binding	Compile-time	Method overloading, static methods
Dynamic Binding	Runtime	Method overriding

- **Abstraction**

59. What is Abstraction in Object-Oriented Programming?

- Hiding internal details and showing only the essential features.
- Achieved through:
  - **Abstract classes**
  - **Interfaces**

---

60. How is Abstraction different from Encapsulation?

Feature	Abstraction	Encapsulation
Purpose	Hide implementation	Protect internal state
Focus	What to do	How to do
Achieved by	Abstract classes and interfaces	Access modifiers (private, public)

---

61. What is an abstract class in Java?

- A class declared using abstract keyword.
- Cannot be instantiated.
- Can have both **abstract and concrete methods**.

---

62. Is it allowed to mark a method as abstract without marking the class abstract?

❌ **No**

- If a method is abstract, the class **must be declared abstract**.

---

63. Is it allowed to mark a method abstract as well as final?

❌ **No**

- final methods cannot be overridden.
- Abstract methods **must be overridden** in subclasses.

---

64. Can we instantiate an abstract class in Java?

❌ **No**

- But we can create a reference and instantiate a subclass.

---

65. What is an interface in Java?

- A blueprint of a class.
- Contains only **abstract methods** (Java 7).
- From **Java 8**, it can have:
  - **Default** methods
  - **Static** methods

---

66. Is it allowed to mark an interface method as static?

✅ **Yes** (from Java 8).

- Static methods belong to the interface and cannot be overridden.

---

67. Why an Interface cannot be marked as final in Java?

- Interfaces are meant to be implemented, so marking them final would make no sense.

---

68. What is a marker interface?

- An interface with **no methods**.
- Used to provide **special behavior** to classes.
  Example: Serializable, Cloneable

---

69. What can we use instead of Marker Interface?

- **Annotations**

---

70. How are Annotations better than Marker Interfaces?

- Annotations provide more flexibility and metadata than marker interfaces.
- Compile-time checking is possible with annotations.

---

71. What is the difference between abstract class and interface in Java?

Feature	Abstract Class	Interface
Methods	Abstract and concrete methods	Abstract (Java 7), Default/Static (Java 8+)
Constructor	Can have a constructor	Cannot have a constructor
Access Modifiers	Can have any modifier	Public and static only
Multiple Inheritance	Not supported	Supported

---

72. Does Java allow private and protected modifiers for variables in interfaces?

❌ **No** – Only public, static, and final are allowed.

---

73. How can we cast an object reference to an interface reference?

✅ **Yes**

```
class Test implements Runnable {
   public void run() {}
}
```

Runnable r = new Test(); // Cast to interface

- **Final**

74. How can you change the value of a final variable in Java?

❌ **You cannot** change the value of a final variable after assignment.

---

75. Can a class be marked final in Java?

✅ **Yes** – A final class cannot be inherited.

Example:

```
final class Test {}
```

---

76. How can we create a final method in Java?

- Use final keyword before the method declaration.
- Final methods **cannot be overridden**.

Example:

```
class Test {
    final void display() {}
}
```

---

77. How can we prohibit inheritance in Java?

- Declare the class as **final**.

---

78. Why is Integer class final in Java?

- To ensure immutability and prevent subclassing, which might break consistency.

---

79. What is a blank final variable in Java?

- A final variable declared but **not initialized**.
- Must be initialized in the **constructor**.

Example:

```
class Test {
    final int x;
    Test(int x) {
        this.x = x;
    }
}
```

---

80. How can we initialize a blank final variable?

- In a **constructor** or **instance block**.
- Cannot be left uninitialized.

81. Is it allowed to declare main method as final?

✅ **Yes** – We can declare the main method as final.

- It will prevent subclasses from overriding it.

Example:

```
public class Test {
    public static final void main(String[] args) {
        System.out.println("Final main method");
    }
}
```

- **Package**

82. What is the purpose of package in Java?

- A package is used to **group related classes** and interfaces.
- Provides **access protection** and **namespace management**.

Example:
package mypackage;
public class MyClass { }

---

83. What is java.lang package?

- It is the **default package** in Java.
- Contains fundamental classes like String, Math, Thread, Exception, etc.

---

84. Which is the most important class in Java?

- Object class – It is the **parent** of all classes in Java.

---

85. Is it mandatory to import java.lang package every time?

❌ **No** – It is imported by default.

---

86. Can you import the same package or class twice in your class?

✅ **Yes** – No compilation error, but the **JVM** will load it only once.

---

87. What is a static import in Java?

- Allows importing **static members** (fields and methods) directly.
- Reduces the need to specify the class name.

Example:
import static java.lang.Math.PI;
System.out.println(PI);

---

88. What is the difference between import static com.test.Fooclass and import com.test.Fooclass?

Import Type	Purpose
import com.test.FooClass	Imports the entire class
import static com.test.FooClass	Imports static members of the class directly

---

- **Internationalization**

89. What is Locale in Java?

- A Locale object represents **geographical, political, and cultural** information.
- Used for formatting **dates**, **numbers**, and **messages**.

---

90. How will you use a specific Locale in Java?

Example:
Locale locale = new Locale("en", "US");
System.out.println(locale.getDisplayCountry()); // Output: United States

---

- **Serialization**

91. What is serialization?

- Serialization is the process of **converting an object into a byte stream**.

---

92. What is the purpose of serialization?

- To **save the state** of an object and recreate it later.
- Used in **file storage** and **network communication**.

---

93. What is deserialization?

- The process of converting a **byte stream back to an object**.

---

94. What is Serialization and Deserialization conceptually?

- **Serialization** → Object → Byte stream
- **Deserialization** → Byte stream → Object

---

95. Why do we mark a data member transient?

- Transient fields are **skipped** during serialization.
  Example:

```
class Test implements Serializable {
   transient int x; // will not be serialized
}
```

---

96. Is it allowed to mark a method as transient?

❌ **No** – Only fields can be marked as transient.

---

97. How does marking a field as transient make it possible to serialize an object?

- It allows the object to be serialized without including sensitive or non-serializable data.

---

98. What is Externalizable interface in Java?

- Provides **custom serialization** using writeExternal() and readExternal() methods.
- More control over the serialization process.

---

99. What is the difference between Serializable and Externalizable interface?

Feature	Serializable	Externalizable
Methods	No methods (handled by JVM)	writeExternal() and readExternal()
Performance	Less control	More control
Marker Interface	Yes	No

- **Reflection**

100. What is Reflection in Java?

- Reflection is the ability to **inspect and modify** classes, methods, and fields **at runtime**.

---

101. What are the uses of Reflection in Java?

- Debugging
- Testing
- Creating objects dynamically

---

102. How can we access a private method of a class from outside the class?

- Using **Reflection**
  Example:

```
class Test {
   private void display() {
      System.out.println("Private method");
```

```
    }
}

Test t = new Test();
Method m = Test.class.getDeclaredMethod("display");
m.setAccessible(true);
m.invoke(t);
```

---

103. How can we create an Object dynamically at Runtime in Java?

- Using Class.forName()
  Example:
  Class<?> cls = Class.forName("Test");
  Object obj = cls.newInstance();

- **Garbage Collection**
  104. What is Garbage Collection in Java?
- Process of **automatically reclaiming unused memory**.
- Done by the **JVM**.

---

105. Why Java provides Garbage Collector?

- To prevent **memory leaks** and **optimize memory usage**.

---

106. What is the purpose of gc() in Java?

- Suggests the JVM to perform **garbage collection**.
  Example:
  System.gc();

---

107. How does Garbage Collection work in Java?

- JVM identifies objects with **no references** and removes them.

---

108. When does an object become eligible for Garbage Collection in Java?

- When it has **no active references**.

---

109. Why do we use finalize() method in Java?

- Called by the garbage collector **before an object is destroyed**.
- Used for **cleanup**.

---

110. What are the different types of References in Java?

- **Strong Reference**
- **Soft Reference**
- **Weak Reference**
- **Phantom Reference**

---

111. How can we reference an unreferenced object again?

- By assigning it to a new reference.

---

112. What kind of process is the Garbage collector thread?

- A **daemon thread** (low priority).

- **Runtime Class**

113. What is the purpose of the Runtime class?

- Provides methods to interact with the **JVM**.

---

114. How can we invoke an external process in Java?
Example:
Runtime.getRuntime().exec("notepad");

---

115. What are the uses of Runtime class?

- Memory management
- Execute processes
- Shutdown hooks

- **Inner Classes**
  116. What is a Nested class?
- A class within another class.

---

117. How many types of Nested classes are in Java?

1. **Static Nested Class**
2. **Non-static Inner Class**
3. **Local Inner Class**
4. **Anonymous Inner Class**

---

118. Why do we use Nested Classes?

- To logically group classes
- To increase encapsulation

---

119. What is the difference between a Nested class and an Inner class in Java?

Type	Nested Class	Inner Class
Static	Yes	No
Instance Reference	No	Yes

---

120. What is a Nested interface?

- An interface defined **inside a class** or another interface.
  Example:

```
class Test {
   interface NestedInterface {
      void display();
   }
}
```

- **Inner Classes**
  121. How can we access the non-final local variable inside a Local Inner class?
- In Java 8 and later, local variables inside a local inner class must be **effectively final** (i.e., their value must not change after being assigned).
- If the variable is not effectively final, the compiler will throw an error.
  Example:

```
class Outer {
   void display() {
      int x = 10; // effectively final
```

```
        class Inner {
            void show() {
                System.out.println(x);
            }
        }
        Inner inner = new Inner();
        inner.show();
    }
}
```

---

122. Can an Interface be defined in a Class?

✅ **Yes** – An interface can be defined inside a class or another interface.

Example:

```
class Outer {
    interface InnerInterface {
        void display();
    }
}
```

---

123. Do we have to explicitly mark a Nested Interface public static?

❌ **No** – By default, nested interfaces are static and public.

---

124. Why do we use Static Nested interface in Java?

- To define a contract within a class without creating an instance of the outer class.

Example:

```
class Outer {
    static interface InnerInterface {
        void display();
    }
}
```

- **String**

125. What is the meaning of Immutable in the context of String class in Java?

- Once a String object is created, its value **cannot be changed**.

---

126. Why a String object is considered immutable in Java?

- String objects are stored in the **String pool**.
- If they were mutable, it could cause **security issues** and **unexpected changes**.

---

127. How many objects does the following code create?

String s = new String("Hello");

- **Two objects** are created:

1. "Hello" → created in the **string pool**.
2. new String() → created on the **heap**.

---

128. How many ways are there in Java to create a String object?

1. Using **String literal**
2. Using **new String()**
3. Using **String.valueOf()**
4. Using **String concatenation**

---

129. How many objects does the following code create?

String s1 = "Hello";
String s2 = "Hello";

- **One object** – Both s1 and s2 refer to the same object in the **string pool**.

---

130. What is String interning?

- The process of storing only **one copy** of each distinct String in the **string pool** to save memory.

---

131. Why Java uses String literal concept?

- To save memory and improve performance by avoiding **duplicate string objects**.

---

132. What is the basic difference between a String and StringBuffer object?

Feature	String	StringBuffer
Mutability	Immutable	Mutable
Performance	Slow	Fast
Thread Safety	Yes	Yes

---

133. How will you create an immutable class in Java?

- Mark the class as **final**.
- Make all fields **private** and **final**.
- Provide **no setter** methods.
- Initialize fields through the **constructor**.

Example:

```
final class MyClass {
   private final int value;

   MyClass(int value) {
      this.value = value;
   }

   public int getValue() {
      return value;
   }
}
```

---

134. What is the use of toString() method in Java?

- Returns the **string representation** of an object.

---

135. Arrange the three classes String, StringBuffer and StringBuilder in the order of efficiency for String processing operations?

- StringBuilder > StringBuffer > String

- **Exception Handling**

136. What is Exception Handling in Java?

- Mechanism to handle **runtime errors** and ensure the **normal flow** of the program.

---

137. In Java, what are the differences between a Checked and Unchecked exception?

Type	Checked	Unchecked
Handled at Compile-time	Yes	No
Inheritance	Extends Exception	Extends RuntimeException

---

138. What is the base class for Error and Exception classes in Java?

- **Throwable**

---

139. What is a finally block in Java?

- A block that **executes always**, even if an exception occurs.

---

140. What is the use of finally block in Java?

- Used for **cleanup operations** like closing files or database connections.

---

141. Can we create a finally block without creating a catch block?

✅ **Yes**

---

142. Do we have to always put a catch block after a try block?

❌ **No** – A try block can be followed by a finally block only.

---

143. In what scenarios, a finally block will not be executed?

- System.exit()
- JVM crash
- Power failure

---

144. Can we re-throw an Exception in Java?

✅ **Yes** – Using the throw keyword.

---

145. What is the difference between throw and throws in Java?

Keyword	Purpose
throw	Used to throw an exception manually
throws	Declares an exception in the method signature

---

146. What is the concept of Exception Propagation?

- If an exception is not caught, it is **propagated** to the calling method.

---

147. When we override a method in a Child class, can we throw an additional Exception that is not thrown by the Parent class method?

❌ **No** – It would break the parent-child relationship.

- **Multi-threading**

148. How Multi-threading works in Java?

- Multiple threads execute concurrently using the **Thread class** or **Runnable interface**.

Example:

```
class MyThread extends Thread {
    public void run() {
        System.out.println("Thread is running");
    }
}
```

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

---

149. What are the advantages of Multithreading?

✅ **Faster execution**
✅ **Efficient CPU utilization**
✅ **Better responsiveness**

---

150. What are the disadvantages of Multithreading?

❌ **Complex debugging**
❌ **Race conditions**
❌ **Deadlocks**

- **Multi-threading**

151. What is a Thread in Java?

- A thread is a **lightweight process** that allows the concurrent execution of tasks.
- Threads in Java are created using:
  - Extending the Thread class
  - Implementing the Runnable interface

---

152. What is a Thread's priority and how is it used in scheduling?

- Each thread is assigned a priority value between **1** (MIN_PRIORITY) to **10** (MAX_PRIORITY).
- The thread scheduler uses priority to decide which thread to execute first.

Example:

```
Thread t = new Thread();
t.setPriority(Thread.MAX_PRIORITY); // Priority set to 10
```

---

153. What are the differences between Pre-emptive Scheduling and Time Slicing Scheduler?

Type	Pre-emptive Scheduling	Time Slicing Scheduler
Execution	Higher priority thread executes first	Threads execute in a round-robin fashion
Control	Controlled by the OS	Controlled by the Java scheduler
Example	Windows OS	JVM

---

154. Is it possible to call run() method instead of start() on a thread in Java?

✅ **Yes**, but it will not create a new thread — it will execute in the same thread as the caller.

Example:

```
Thread t = new Thread(() -> System.out.println("Running"));
t.run(); // No new thread created
```

---

155. How will you make a user thread into a daemon thread if it has already started?

❌ **Not possible** — A thread must be marked as a daemon before calling start().

---

156. Can we start a thread two times in Java?

❌ **No** — It will throw an IllegalThreadStateException.

Example:

```
Thread t = new Thread();
```

```
t.start();
t.start(); // Throws exception
```

---

157. In what scenarios can we interrupt a thread?

- When a thread is in **sleep**, **wait**, or **blocked** state.

---

158. In Java, is it possible to lock an object for exclusive use by a thread?

✅ **Yes** — By using the synchronized keyword.

Example:

```
synchronized(obj) {
   // Critical section
}
```

---

159. How is notify() method different from notifyAll() method?

Method	Behavior
notify()	Wakes up **one** waiting thread
notifyAll()	Wakes up **all** waiting threads

- **Collections**

160. What are the differences between a Vector and an ArrayList?

Feature	Vector	ArrayList
Thread-safe	Yes	No
Performance	Slower	Faster
Default size increment	Doubles the size	Increases by 50%

---

161. What are the differences between Collection and Collections in Java?

Term	Description
Collection	Interface for collection types (List, Set, Queue)
Collections	Utility class with static methods for collection manipulation

---

162. In which scenario is LinkedList better than ArrayList in Java?

- When **frequent insertions and deletions** are required (LinkedList uses a doubly linked list).

---

163. What are the differences between a List and Set collection in Java?

Feature	List	Set
Allows duplicates	Yes	No
Order	Maintains insertion order	No guaranteed order

---

164. What are the differences between a HashSet and TreeSet collection in Java?

Feature	HashSet	TreeSet
Order	No order	Sorted order
Null values	Allows one null value	Does not allow null

---

165. How will you decide when to use a List, Set, or Map collection?

Collection Use Case

Collection	Use Case
List	When order and duplicates are important
Set	When uniqueness is required
Map	When key-value pairs are required

---

166. What are the differences between a HashMap and a Hashtable in Java?

Feature	HashMap	Hashtable
Thread-safe	No	Yes
Null values	Allows null keys and values	Does not allow null
Performance	Faster	Slower

---

167. What are the differences between a HashMap and a TreeMap?

Feature	HashMap	TreeMap
Order	No order	Sorted order
Null keys	Allows one null key	No null keys allowed

---

168. What are the differences between Comparable and Comparator?

Feature	Comparable	Comparator
Method	compareTo()	compare()
Implementation	Class implements Comparable	Separate class implements Comparator

---

169. What is the purpose of a Properties file?

- Used to store **key-value pairs** for **configuration**.

---

170. What is the reason for overriding equals() method?

- To define **custom logic** for object comparison.

---

171. How does hashCode() method work in Java?

- Returns an integer that represents the object's memory location.

---

172. Is it a good idea to use Generics in collections?

✅ **Yes** – Ensures **type safety** and avoids **ClassCastException**.

- **Mixed Questions**

173. What are Wrapper classes in Java?

- Classes that convert **primitive types** into **objects** (e.g., Integer, Double).

---

174. What is the purpose of the native method in Java?

- To use **non-Java code** (e.g., C or C++).

---

175. What is the System class?

- A **final class** that provides access to system properties and utilities.

---

176. What is System, out, and println in System.out.println method call?

- System – Class
- out – Static PrintStream object

- println – Method of PrintStream

---

177. What is the other name of Shallow Copy in Java?

- **Bitwise Copy**

---

178. What is the difference between Shallow Copy and Deep Copy in Java?

Feature	Shallow Copy	Deep Copy
References	Copies object references	Copies actual objects
Impact on changes	Changes reflect in both copies	Changes don't reflect

---

179. What is a Singleton class?

- A class that allows only **one instance** at a time.

Example:

```
class Singleton {
   private static Singleton instance = new Singleton();
   private Singleton() {}
   public static Singleton getInstance() {
      return instance;
   }
}
```

---

180. What is the difference between Singleton class and Static class?

Feature	Singleton Class	Static Class
Instance	Single instance	No instance
Inheritance	Can inherit	Cannot inherit

- **Collections Framework**

181. What is the difference between Collection and Collections Framework in Java?

Term	Description
Collection	Interface that represents a group of objects (List, Set, Queue)
Collections Framework	Complete architecture of collection classes and interfaces in Java (List, Set, Map, etc.)

---

182. What are the main benefits of Collections Framework in Java?

- ✅ **Reusable** – Common data structures can be reused.
- ✅ **Performance** – Well-optimized classes and interfaces.
- ✅ **Consistency** – Unified interface for different collection types.
- ✅ **Convenience** – Utility methods for sorting, searching, and synchronization.

---

183. What is the root interface of Collection hierarchy in Java?

- **Collection** interface is the root.
- **Iterable** is the super interface of Collection.

---

184. What are the main differences between Collection and Collections?

Term	Description
Collection	Interface representing a group of objects

Term	Description
Collections	Utility class with static methods for working with collections

---

185. What are the Thread-safe classes in Java Collections framework?

- **Vector**
- **Hashtable**
- **Stack**
- **synchronizedList(), synchronizedSet(), synchronizedMap()** from Collections

---

186. How will you efficiently remove elements while iterating a Collection?

- Use **Iterator** with remove() method.

Example:

```
Iterator<Integer> it = list.iterator();
while (it.hasNext()) {
   if (it.next() < 5) {
      it.remove();
   }
}
```

---

187. How will you convert a List into an array of integers like int[]?

```
int[] arr = list.stream().mapToInt(i -> i).toArray();
```

---

188. How will you convert an array of primitive integers int[] to a List collection?

```
List<Integer> list = Arrays.stream(arr).boxed().collect(Collectors.toList());
```

---

189. How will you run a filter on a Collection?

```
list.stream().filter(x -> x > 10).forEach(System.out::println);
```

---

190. How will you convert a List to a Set?

```
Set<Integer> set = new HashSet<>(list);
```

---

191. How will you remove duplicate elements from an ArrayList?

```
list = new ArrayList<>(new HashSet<>(list));
```

---

192. How can you maintain a Collection with elements in sorted order?

- Use **TreeSet** or **TreeMap**.
- Use Collections.sort() for List.

---

193. What is the difference between Collections.emptyList() and creating a new instance of Collection?

- Collections.emptyList() creates an **immutable empty list**.
- new ArrayList<>() creates a **mutable empty list**.

---

194. How will you copy elements from a Source List to another list?

```
Collections.copy(destList, sourceList);
```

---

195. What are the Java Collection classes that implement List interface?

- **ArrayList**
- **LinkedList**
- **Vector**

- **Stack**

---

196. What are the Java Collection classes that implement Set interface?

- **HashSet**
- **LinkedHashSet**
- **TreeSet**

---

197. What is the difference between an Iterator and ListIterator in Java?

Feature	Iterator	ListIterator
Direction	Forward only	Both forward and backward
Index-based	No	Yes
Add/Modify	No	Yes

---

198. What is the difference between Iterator and Enumeration?

Feature	Iterator	Enumeration
Methods	hasNext(), next(), remove()	hasMoreElements(), nextElement()
Removal	Yes	No
Scope	Modern	Legacy (before Java 2)

---

199. What is the difference between an ArrayList and a LinkedList?

Feature	ArrayList	LinkedList
Memory	Contiguous memory	Node-based
Access	Fast (O(1))	Slow (O(n))
Insertion/Deletion	Slow	Fast

---

200. What is the difference between a Set and a Map?

Feature	Set	Map
Key-Value	No	Yes
Duplicates	No	No duplicate keys

---

201. What is the use of a Dictionary class?

- Legacy class (before Map) for key-value pairs.

---

202. What is the default size of load factor in a HashMap collection in Java?

- **0.75**

---

203. What is the significance of load factor in a HashMap in Java?

- Determines when the map resizes to prevent collisions.

---

204. What is a Hash Collision? How Java handles hash collision in HashMap?

- When two keys generate the same hash value.
- Java handles it using a **LinkedList** or **Balanced Tree** (Java 8+).

---

205. What are the Hash Collision resolution techniques?

✅ **Chaining** – Store colliding elements in a linked list.
✅ **Open Addressing** – Find next available slot.

---

206. What is the difference between Queue and Stack?

Feature	Queue	Stack
Order	FIFO	LIFO
Methods	add(), poll(), peek()	push(), pop(), peek()

---

207. What is an Iterator in Java?

- Interface used to **iterate over collections**.

---

208. What is the design pattern used in the implementation of Enumeration in Java?

- **Iterator Pattern**

---

209. Which methods do we need to override to use an object as key in a HashMap?

- equals()
- hashCode()

---

210. How will you reverse a List in Java?

Collections.reverse(list);

---

211. How will you convert an array of String objects into a List?

List<String> list = Arrays.asList(array);

---

212. What is the difference between peek(), poll(), and remove() methods of Queue interface?

Method	Description
peek()	Returns the head without removing
poll()	Returns the head and removes it
remove()	Same as poll() but throws exception if queue is empty

---

213. What is the difference between Array and ArrayList?

Feature	Array	ArrayList
Size	Fixed	Dynamic
Primitive types	Yes	No

---

214. What are the main differences between HashMap and ConcurrentHashMap?

Feature	HashMap	ConcurrentHashMap
Thread-safe	No	Yes
Performance	Fast	Slower

---

215. What is CopyOnWriteArrayList? How it is different from ArrayList?

- CopyOnWriteArrayList creates a new copy of the list for modification.
- **Thread-safe**.

---

216. How does ConcurrentHashMap work?

- Uses **segments** to allow multiple threads to access different segments concurrently.

---

217. What is an EnumSet in Java?

- Specialized set for **enumerations**.

---

218. How can you make a Collection class read-only in Java?
Collections.unmodifiableList(list);

- **Collections Framework (Continued)**
  219. What are the different ways to iterate elements of a list in Java?
  ✅ **Using for loop:**

```
for (int i = 0; i < list.size(); i++) {
    System.out.println(list.get(i));
}
```

  ✅ **Using enhanced for loop:**

```
for (String s : list) {
    System.out.println(s);
}
```

  ✅ **Using Iterator:**

```
Iterator<String> it = list.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}
```

  ✅ **Using ListIterator:**

```
ListIterator<String> listIterator = list.listIterator();
while (listIterator.hasNext()) {
    System.out.println(listIterator.next());
}
```

  ✅ **Using Streams:**

```
list.stream().forEach(System.out::println);
```

  ✅ **Using forEach method:**

```
list.forEach(System.out::println);
```

---

220. What is CopyOnWriteArrayList? How is it different from ArrayList in Java?

Feature	ArrayList	CopyOnWriteArrayList
Thread Safety	Not thread-safe	Thread-safe
Modification	Modifies directly	Creates a new copy on modification
Performance	Faster (better for read-heavy)	Slower (due to copy on write)
Use Case	General use	Multi-threaded scenarios with more reads

---

221. How is remove() method implemented in a HashMap?

1. Calculate **hash code** of the key.
2. Find the bucket using hash % capacity.
3. If multiple entries exist (due to collision):
   - Search the linked list (or tree).
   - Remove the matched node.

---

222. What is BlockingQueue in Java Collections?

- BlockingQueue is part of the **java.util.concurrent** package.
- Supports **producer-consumer** scenarios.
- Blocks the thread when:

- take() → If the queue is empty.
- put() → If the queue is full.
  Example:
  BlockingQueue<Integer> queue = new LinkedBlockingQueue<>(5);
  queue.put(1);
  System.out.println(queue.take());

---

223. How is TreeMap class implemented in Java?

- Implemented using **Red-Black Tree**.
- Maintains elements in **sorted order** based on the natural order or a comparator.

---

224. What is the difference between Fail-fast and Fail-safe iterator in Java?

Type	Description	Example
Fail-fast	Throws ConcurrentModificationException if modified during iteration.	ArrayList, HashMap
Fail-safe	Works on a cloned copy, does not throw exception.	ConcurrentHashMap, CopyOnWriteArrayList

---

225. How does ConcurrentHashMap work in Java?

- Divides the map into **segments**.
- Allows multiple threads to modify different segments concurrently.
- No locking on reads.
- Uses **CAS (Compare-And-Swap)** to ensure thread safety.

---

226. What is the importance of hashCode() and equals() methods?

- Ensures correct behavior of **HashMap**, **HashSet**, and other hash-based collections.
- hashCode() → Determines the bucket.
- equals() → Confirms object equality.

---

227. What is the contract of hashCode() and equals() methods in Java?

✅ If equals() returns true, hashCode() should return the **same value**.
✅ If hashCode() is the same, equals() may return **false**.

---

228. What is an EnumSet in Java?

- Specialized set for **Enum types**.
- Uses a **bit-vector** for high performance.
  Example:
  EnumSet<Day> days = EnumSet.of(Day.MONDAY, Day.FRIDAY);

---

229. What are the main Concurrent Collection classes in Java?

- ConcurrentHashMap
- CopyOnWriteArrayList
- CopyOnWriteArraySet
- BlockingQueue
- ConcurrentLinkedQueue

---

230. How will you convert a Collection to SynchronizedCollection in Java?
List<String> list = Collections.synchronizedList(new ArrayList<>());

---

231. How is IdentityHashMap different from a regular Map in Java?

- Uses **reference equality (==)** instead of equals() for key comparison.

---

232. What is the main use of IdentityHashMap?

- When key comparison by **reference** is required (e.g., object identity).

---

233. How can we improve the performance of IdentityHashMap?

- Reduce the number of collisions by setting an appropriate **initial capacity**.

---

234. Is IdentityHashMap thread-safe?

- **No** — It's not thread-safe.

---

235. What is a WeakHashMap in Java?

- Keys are stored as **WeakReferences**.
- Garbage collector removes keys when they are no longer referenced elsewhere.

---

236. How can you make a Collection class read-only in Java?

List<String> list = Collections.unmodifiableList(new ArrayList<>());

---

237. When is UnsupportedOperationException thrown in Java?

- When modifying an **unmodifiable** collection.
  Example:
  List<String> list = Collections.unmodifiableList(new ArrayList<>());
  list.add("Test"); // Throws UnsupportedOperationException

---

238. How to sort a list of Customer objects by firstName attribute?

list.sort(Comparator.comparing(Customer::getFirstName));

---

239. What is the difference between Synchronized Collection and Concurrent Collection?

Type	Description
Synchronized	Entire collection is locked
Concurrent	Only part of collection is locked

---

240. What is the scenario to use ConcurrentHashMap in Java?

- High-concurrency and high-read scenarios.

---

241. How will you create an empty Map in Java?

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

---

242. What is the difference between remove() method of Collection and remove() method of Iterator?

Method	Description
Collection.remove()	Removes element by value
Iterator.remove()	Removes last element returned by next()

---

243. Between an Array and ArrayList, which is preferred for storing objects?

- **ArrayList** – Dynamic size, easier to manage.

---

244. Is it possible to replace Hashtable with ConcurrentHashMap in Java?

- **Yes** – ConcurrentHashMap is more efficient.

---

245. How is CopyOnWriteArrayList different from ArrayList and Vector?

- Thread-safe with **copy on modification**.
- Slower write operations, but faster reads.

---

246. Why does ListIterator have an add() method but Iterator does not?

- ListIterator allows **bidirectional** navigation and modification.

---

247. Why do we sometimes get ConcurrentModificationException during iteration?

- When modifying a collection while iterating using a **fail-fast** iterator.

---

248. How will you convert a Map to a List in Java?

List<Integer> list = new ArrayList<>(map.values());

---

249. How can we create a Map with reverse view and lookup in Java?

- Use **BiMap** from Google Guava.

---

250. How will you create a shallow copy of a Map?

Map<String, String> copy = new HashMap<>(originalMap);

1. **Java Collections and Multithreading (Continued)**

251. Why we cannot create a generic array in Java?

- Due to **type erasure**, the type information of generics is removed at runtime.
- Arrays have **reified type** (type information is retained at runtime).
- This creates a conflict between generics and arrays.

Example:

```
// Compilation error
List<String>[] array = new ArrayList<String>[10];
```

---

252. What is a PriorityQueue in Java?

- A PriorityQueue is a **queue** where elements are ordered based on their **natural order** or a provided **comparator**.
- Head of the queue is the **least element** (for min-heap).

Example:

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(3);
pq.add(1);
pq.add(2);
System.out.println(pq.poll()); // Output: 1 (smallest element)
```

---

253. Important points while using Java Collections Framework

- ✅ Use **Generics** to avoid runtime errors.
- ✅ Use **immutable collections** when modification is not needed.
- ✅ Prefer **ArrayList** for random access; use **LinkedList** for insert/delete.
- ✅ For thread safety, use **concurrent collections**.
- ✅ Use **Collections utility** methods for sorting, searching, etc.

---

254. How to pass a Collection as an argument and ensure it's not modified?

Use **Collections.unmodifiableList()**:

```
void processList(List<String> list) {
   list.add("new"); // Throws UnsupportedOperationException
}

List<String> list = Collections.unmodifiableList(new ArrayList<>());
processList(list);
```

---

255. How does HashMap work in Java?

2. hashCode() → Determines the bucket location.
3. If collision occurs → Store as a linked list (or red-black tree in Java 8+).
4. equals() → Confirms the correct key-value pair.

---

256. How is HashSet implemented in Java?

- Backed by a **HashMap**.
- The elements are stored as keys in the map, and the values are a constant object (PRESENT).
  Example:

```
Set<String> set = new HashSet<>();
set.add("A");
```

---

257. What is a NavigableMap in Java?

- Extends SortedMap.
- Provides navigation methods like **lowerKey(), floorKey(), ceilingKey(), higherKey()**.

---

258. Difference between descendingKeySet() and descendingMap()?

Method	Description
descendingKeySet()	Returns a set of keys in reverse order.
descendingMap()	Returns the map itself in reverse order.

---

259. Advantage of NavigableMap over Map

- Additional navigation methods like lowerKey() and floorKey() make it easier to search elements.

---

260. Difference between headMap(), tailMap() and subMap() in NavigableMap

Method	Description
headMap(K toKey)	Returns keys less than toKey.
tailMap(K fromKey)	Returns keys greater than or equal to fromKey.
subMap(K fromKey, K toKey)	Returns keys between fromKey (inclusive) and toKey (exclusive).

---

261. How to sort objects by natural order in a List?

```
list.sort(Comparator.naturalOrder());
```

---

262. How to get a Stream from a List in Java?

```
List<String> list = Arrays.asList("A", "B", "C");
list.stream().forEach(System.out::println);
```

---

263. Can we get a Map from a Stream in Java?

Yes, using **Collectors.toMap()**:
Map<String, Integer> map = list.stream()
.collect(Collectors.toMap(s -> s, String::length));

---

264. Popular implementations of Deque in Java

- ArrayDeque
- LinkedList

Example:
Deque<String> deque = new ArrayDeque<>();
deque.add("A");
deque.addFirst("B");
deque.addLast("C");

---

5. Advanced Multi-threading

265. What is a Thread in Java?

- A thread is a **lightweight process**.
- Each thread has its own **call stack**.

---

266. What is the priority of a Thread and how is it used in scheduling?

- Determines how frequently a thread gets CPU time.
- **Higher priority** → More CPU time.

---

267. What is the default priority of a Thread in Java?

- Thread.NORM_PRIORITY = 5

---

268. Three priorities that can be set on a Thread in Java

- Thread.MIN_PRIORITY = 1
- Thread.NORM_PRIORITY = 5
- Thread.MAX_PRIORITY = 10

---

269. What is the purpose of join() method in Thread class?

- Makes the calling thread **wait** for the target thread to complete.

Example:
Thread t = new Thread(() -> System.out.println("Running"));
t.start();
t.join(); // Main thread will wait until t finishes.

---

270. Difference between wait() and sleep()

Method	Description
wait()	Releases lock and waits (used with synchronized).
sleep()	Pauses execution for a fixed time without releasing the lock.

---

271. Is it possible to call run() instead of start() on a thread?

- Yes, but it will run in the **main thread** instead of starting a new thread.

---

272. What is a Daemon thread in Java?

- A thread that runs in the **background** and exits when all user threads complete.

---

273. How to make a regular thread a Daemon thread?

```
Thread t = new Thread();
t.setDaemon(true);
```

---

274. How to make a user thread into daemon thread if it has already started?

- **Impossible** – A thread cannot be converted into a daemon once started.

---

275. Can we start a thread two times in Java?

- **No** – Once a thread has terminated, it cannot be restarted.

---

276. What is a Shutdown hook in Java?

- A thread that executes **just before the JVM shuts down**.
- Used for cleanup.
  Example:
  Runtime.getRuntime().addShutdownHook(new Thread(() -> System.out.println("Shutdown hook running")));

---

277. What is synchronization in Java?

- Synchronization allows only **one thread** to access a resource at a time.
  Example:

```
synchronized void display() {
   // Code here
}
```

---

278. What is the purpose of Synchronized block in Java?

- Limits the lock to a **specific block** instead of the entire method.
  Example:

```
void display() {
   synchronized (this) {
      // Code here
   }
}
```

---

279. What is static synchronization?

- Synchronizes at the **class level** instead of object level.
  Example:

```
synchronized static void display() {
   // Code here
}
```

---

280. What is a Deadlock situation?

- When two or more threads are waiting for each other to release a lock, causing an **infinite block**.
  Example:

```
synchronized void method1() {
   synchronized (obj2) { }
}

synchronized void method2() {
   synchronized (obj1) { }
}
```

- **Advanced Multi-threading in Java**

281. What is the meaning of concurrency?

- **Concurrency** means executing multiple tasks **simultaneously**.
- In Java, concurrency is handled using **threads**.
- Tasks may run on:
  - **Single processor** (time-sliced)
  - **Multiple processors** (true parallelism)

---

282. What is the main difference between process and thread?

Aspect	Process	Thread
Definition	Independent execution unit with its own memory	Lightweight execution unit within a process
Memory Sharing	Does **not share memory** with other processes	Shares memory with other threads of the same process
Context Switching	Expensive (involves OS operations)	Cheaper (less overhead)
Creation Time	High	Low

---

283. What is a process and thread in the context of Java?

- **Process** – An executing instance of a Java application.
- **Thread** – A lightweight subprocess within a process.

Example:

```
class MyThread extends Thread {
   public void run() {
      System.out.println("Thread is running");
   }
}

MyThread t = new MyThread();
t.start(); // Creates a new thread
```

---

284. What is a Scheduler?

- Scheduler is part of the **JVM** and **OS**.
- It decides which thread gets CPU time based on:
  - Priority
  - State
  - Scheduling policy

---

285. What is the minimum number of Threads in a Java program?

- Minimum **two threads**:

1. **Main thread**
2. **Garbage Collector thread**

---

286. Properties of a Java thread

✅ Has a **name**

✅ Has a **priority**

✅ Has a **state** (NEW, RUNNABLE, BLOCKED, WAITING, etc.)
✅ Can be **daemon** or **user**

---

287. Different states of a Thread in Java

State	Description
NEW	Created but not yet started
RUNNABLE	Ready to run but waiting for CPU
BLOCKED	Waiting for a monitor lock
WAITING	Waiting indefinitely for another thread
TIMED_WAITING	Waiting for a specific time
TERMINATED	Completed execution

---

288. How to set the priority of a thread in Java?
Use setPriority() method:
t.setPriority(Thread.MAX_PRIORITY);

---

289. Purpose of Thread Groups in Java

- Organize threads into **groups** for easier management.
- Can set priority and manage interruptions collectively.

---

290. Why should we not stop a thread by calling its stop() method?

- stop() is **unsafe** – Can leave shared resources in an inconsistent state.
- Use **interrupt()** instead.

---

291. How to create a Thread in Java?

1. **Extend Thread class**
2. **Implement Runnable interface**
3. **Use Executor framework**

Example using Runnable:

```
class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Thread is running");
    }
}
Thread t = new Thread(new MyRunnable());
t.start();
```

---

292. How to stop a thread in the middle of execution in Java?

- Use interrupt() method:

```
if (Thread.interrupted()) {
    return;
}
```

---

293. How to access the current thread in a Java program?
Use Thread.currentThread() method:

```
Thread t = Thread.currentThread();
System.out.println(t.getName());
```

---

294. What is Busy Waiting in Multi-threading?

- A thread continuously checks a condition without releasing CPU time.
- Inefficient → Wastes CPU cycles.

---

295. How to prevent busy waiting in Java?

- Use wait() or sleep() instead of looping:

```
synchronized (obj) {
    obj.wait();
}
```

---

296. Can we use Thread.sleep() for real-time processing in Java?

- No – sleep() is not precise for real-time requirements.
- Depends on OS scheduling.

---

297. Can we wake up a thread that has been put to sleep using Thread.sleep()?

- No – sleep() is not interruptible.

---

298. Two ways to check if a Thread has been interrupted

1. isInterrupted()
2. Thread.interrupted()

---

299. How to make sure the parent thread waits for termination of child thread?

Use join() method:

```
t.join();
```

---

300. How to handle InterruptedException in Java?

Use try-catch block:

```
try {
    Thread.sleep(1000);
} catch (InterruptedException e) {
    Thread.currentThread().interrupt(); // Restore interrupted state
}
```

---

301. Which intrinsic lock is acquired by a synchronized method in Java?

- Acquires the **monitor lock** of the object.

---

302. Can we mark a constructor as synchronized in Java?

- No – Only methods can be synchronized.

---

303. Can we use primitive values for intrinsic locks?

- No – Only objects can be locked.

---

304. Do intrinsic locks have a re-entrant property?

- Yes – A thread can reacquire the same lock it already holds.

---

305. What is an atomic operation?

- An operation that completes **entirely** or **not at all** without interruption.

---

306. Is i++ an atomic operation in Java?

- No – i++ involves:

1. Read
2. Increment
3. Write

---

307. Atomic operations in Java

✅ AtomicInteger
✅ AtomicLong
✅ AtomicBoolean

Example:
AtomicInteger count = new AtomicInteger(0);
count.incrementAndGet();

---

308. Is the following code thread-safe?

No – Use synchronization or atomic classes:
int value = 0;
value++;

---

309. Minimum requirements for a Deadlock

✅ Mutual Exclusion
✅ Hold and Wait
✅ No Preemption
✅ Circular Wait

---

310. How to prevent a Deadlock?

✅ Use a timeout
✅ Acquire locks in a fixed order
✅ Avoid nested locks

---

311. How to detect a Deadlock?

- Use **thread dump** (jstack)
- Use a monitoring tool like **JConsole**

---

312. What is a Livelock?

- Threads keep changing state but **no progress** is made.

---

313. What is Thread Starvation?

- A low-priority thread never gets CPU time due to high-priority threads.

---

314. How can a synchronized block cause Thread Starvation?

- If a high-priority thread keeps acquiring the lock, lower-priority threads will starve.

---

315. What is a Race Condition?

- Two or more threads modify a shared resource at the same time, causing unpredictable results.

---

316. What is a Fair Lock in Multi-threading?

- Ensures that threads are granted access in the order they requested it.

---

317. Which two methods of Object class can implement Producer-Consumer?

- wait() and notify()

---

318. How does JVM determine which thread to wake up on notify()?

- Randomly – Based on **OS scheduling**.

---

319. Is the following code thread-safe for retrieving from a Queue?

- No – Use ConcurrentLinkedQueue for thread-safety.

---

320. How to check if a thread holds a monitor lock on an object?

Use Thread.holdsLock(obj):

```
if (Thread.holdsLock(obj)) {
    System.out.println("Thread holds the lock");
}
```

1. **Advanced Multi-threading in Java (Continued)**

321. What is the use of yield() method in Thread class?

- Thread.yield() allows the current thread to:
  - ✅ Pause its execution
  - ✅ Give other threads a chance to execute
- Scheduler may **ignore** the request.

Example:

```
Thread.yield();
```

---

322. What is an important point to consider while passing an object from one thread to another thread?

- Ensure **thread safety** by:
  - ✅ Using **immutable objects**
  - ✅ Using **synchronization**
  - ✅ Using **concurrent collections**

---

323. What are the rules for creating Immutable Objects?

2. Declare class as final
3. Declare fields as private final
4. No setter methods
5. Initialize fields in the constructor
6. Return deep copies instead of actual objects

Example:

```
final class ImmutableClass {
    private final int value;
    public ImmutableClass(int value) {
        this.value = value;
    }
    public int getValue() {
        return value;
    }
}
```

---

324. What is the use of ThreadLocal class?

- Provides **thread-local variables**.
- Each thread has its own **independent copy** of the variable.

Example:
ThreadLocal<Integer> threadLocal = ThreadLocal.withInitial(() -> 0);
threadLocal.set(100);
System.out.println(threadLocal.get());

---

325. What are the scenarios suitable for using ThreadLocal class?

- ✅ Database connection pooling
- ✅ User session handling
- ✅ Thread-specific data caching

---

326. How to improve the performance of an application by multi-threading?

- ✅ Minimize lock contention
- ✅ Use ReadWriteLock for read-heavy operations
- ✅ Use thread pooling
- ✅ Reduce context switching

---

327. What is scalability in a Software program?

- Ability to handle **increased load** by:
  - ✅ Adding more resources (vertical scaling)
  - ✅ Increasing number of nodes (horizontal scaling)

---

328. How to calculate the maximum speedup of an application using multiple processors?

Use **Amdahl's Law**:
S=1(1−P)+PNS = \frac{1}{(1 - P) + \frac{P}{N}}
Where:

- P = Parallel portion of the program
- N = Number of processors

---

329. What is Lock Contention in multi-threading?

- Occurs when:
  - ✅ Multiple threads compete for the same lock
  - ✅ Reduces performance due to waiting

---

330. Techniques to reduce Lock Contention

- ✅ Minimize critical section size
- ✅ Use ReadWriteLock instead of synchronized
- ✅ Use ConcurrentHashMap instead of HashMap

---

331. Technique to reduce Lock Contention in the following code

- Replace synchronized with ReadWriteLock or Atomic classes:

```
synchronized (obj) {
    // critical section
}
```

---

332. What is Lock Splitting technique?

- Split a single lock into **multiple finer locks**.
- Reduces contention by reducing the lock's granularity.

Example:

- Use **separate locks** for read and write.

---

333. Technique used in ReadWriteLock class for reducing Lock Contention

- Uses **two locks**:
  - ✅ Read Lock → For multiple threads reading simultaneously
  - ✅ Write Lock → For exclusive access while writing

---

334. What is Lock Striping?

- Create **multiple locks** for independent data portions.
- Example → ConcurrentHashMap uses lock striping.

---

335. What is a CAS (Compare-And-Swap) operation?

- **Atomic operation** to update a value only if it matches an expected value.
- No need for locks → Improves performance.

Example:

```
AtomicInteger value = new AtomicInteger(0);
value.compareAndSet(0, 10); // If value is 0, set to 10
```

---

336. Which Java classes use CAS operation?

- ✅ AtomicInteger
- ✅ AtomicLong
- ✅ AtomicReference

---

337. Is object pooling always effective in multi-threading?

- **Not always** – Can cause:
  - ❌ Increased memory usage
  - ❌ Lock contention

---

338. How single-threaded performance improvements can degrade multi-threaded performance

- Caching → Can cause contention in multi-threaded environments.
- Large objects → Can increase memory pressure.

---

339. Relation between Executor and ExecutorService interface

- Executor – Basic interface for executing tasks.
- ExecutorService – Extends Executor with additional methods like:
  - ✅ shutdown()
  - ✅ invokeAll()

---

340. What happens if the queue is full when calling submit() on an ExecutorService?

- Throws **RejectedExecutionException** if queue is full.

---

341. What is a ScheduledExecutorService?

- Executes tasks:
  - ✅ After a delay
  - ✅ At fixed intervals

Example:

```
ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(1);
scheduler.schedule(() -> System.out.println("Delayed task"), 2, TimeUnit.SECONDS);
```

---

342. How to create a Thread pool in Java?

Use Executors:

```
ExecutorService pool = Executors.newFixedThreadPool(4);
```

---

343. Difference between Runnable and Callable interface

Aspect	Runnable	Callable
Return Type	void	V (Generic)
Exception Handling	Cannot throw checked exceptions	Can throw checked exceptions
Method	run()	call()

Example with Callable:

```
Callable<Integer> task = () -> 42;
Future<Integer> future = executor.submit(task);
```

---

344. Uses of Future interface in Java

- ✅ Check if task is complete
- ✅ Get result of task
- ✅ Cancel task

---

345. Difference in concurrency between HashMap and Hashtable

Aspect	HashMap	Hashtable
Thread-safe	No	Yes (Synchronized)
Performance	Faster	Slower
Null Keys/Values	Allows one null key and multiple null values	Does not allow null keys or values

---

346. How to create a synchronized instance of List or Map collection?

Use Collections.synchronizedList() or Collections.synchronizedMap()

```
List<Integer> list = Collections.synchronizedList(new ArrayList<>());
```

---

347. What is a Semaphore in Java?

- Controls access to a shared resource using **permits**.
- Example: Limit number of database connections.

Example:

```
Semaphore semaphore = new Semaphore(2);
semaphore.acquire(); // Get permit
semaphore.release(); // Release permit
```

---

348. What is a CountDownLatch in Java?

- Allows one or more threads to wait until a set of operations are complete.

Example:

```
CountDownLatch latch = new CountDownLatch(3);
latch.countDown(); // Reduce count
latch.await(); // Wait until count reaches zero
```

---

349. Difference between CountDownLatch and CyclicBarrier

Aspect	CountDownLatch	CyclicBarrier
Reusable	No	Yes
Purpose	Wait for events	Synchronize threads

---

350. Scenarios suitable for using Fork/Join framework

✅ Recursive tasks (Divide and Conquer)
✅ Large data processing
✅ Parallel sorting

Example:

```
ForkJoinPool pool = new ForkJoinPool();
pool.invoke(new MyTask());
```

2. Advanced Java Multi-threading and Java 8 Features

351. Difference between RecursiveTask and RecursiveAction class?

Class	Purpose	Return Type	Example
RecursiveTask<V>	Used when the task **returns a result**	Returns V	invoke() or fork() returns result
RecursiveAction	Used when the task **does not return a result**	void	invoke() or fork() does not return a result

Example with RecursiveTask:

```
class MyTask extends RecursiveTask<Integer> {
   protected Integer compute() {
      return 42;
   }
}
```

Example with RecursiveAction:

```
class MyAction extends RecursiveAction {
   protected void compute() {
      System.out.println("Action performed");
   }
}
```

---

352. Can we process stream operations with a Thread pool in Java 8?

- **No** – Parallel streams use the **ForkJoinPool** internally.
- To use a custom thread pool, convert stream to Spliterator and manually execute using ExecutorService.

Example using ExecutorService:

```
ExecutorService executor = Executors.newFixedThreadPool(4);
executor.submit(() -> list.parallelStream().forEach(System.out::println));
```

---

353. Scenarios to use Parallel Stream in Java 8

✅ Large data sets
✅ CPU-intensive tasks
✅ Tasks without shared state or side effects

Example:

```
list.parallelStream().forEach(System.out::println);
```

---

354. How Stack and Heap work in Java multi-threading environment?

✅ **Stack** – Each thread gets its own stack for method calls and local variables.

✅ **Heap** – Shared among all threads; stores objects and class metadata.

---

355. How to take a Thread dump in Java?

1. Use **jstack** tool:
   jstack <PID>
3. Use **kill -3** (Linux).
4. Use **Java VisualVM** tool.

---

356. Which parameter controls the stack size of a thread in Java?

✅ -Xss JVM parameter.

Example:

java -Xss512k MyApp

---

357. How to ensure that two threads T1 and T2 run in sequence in Java?

1. Use join() method:

```
t1.start();
t1.join(); // Wait for T1 to finish
t2.start();
```

2. Use CountDownLatch:

```
CountDownLatch latch = new CountDownLatch(1);
new Thread(() -> {
    // T1 code
    latch.countDown();
}).start();

new Thread(() -> {
    try {
        latch.await(); // Wait for T1
        // T2 code
    } catch (InterruptedException e) { }
}).start();
```

---

5. Java 8 Features

358. New features released in Java 8

✅ Lambda Expressions

✅ Functional Interfaces

✅ Stream API

✅ Default and Static Methods in Interfaces

✅ Optional Class

✅ Method References

✅ Date and Time API

---

359. Main benefits of new features in Java 8

✅ Improved code readability

✅ Enhanced performance

✅ Functional-style programming
✅ Better concurrency handling

---

360. What is a Lambda expression in Java 8?

- Short syntax to define **anonymous functions**.
- Example:
  (int a, int b) -> a + b;

---

361. Three main parts of a Lambda expression

1. **Parameters** → (int a, int b)
2. **Arrow operator** → ->
3. **Body** → a + b

---

362. What is the data type of a Lambda expression?

- Data type → **Functional Interface**
  Example:
  BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;

---

363. Meaning of the following Lambda expression

(a, b) -> a + b

- Takes two parameters (a, b)
- Returns sum of a and b

---

364. Why did Oracle release a new version like Java 8?

✅ Improve productivity
✅ Introduce functional programming
✅ Handle multi-core processing

---

365. Advantages of Lambda expression

✅ Concise and readable
✅ Reduced boilerplate code
✅ Improves parallel processing

---

366. What is a Functional Interface in Java 8?

- An interface with **exactly one abstract method**.
- Example:

```
@FunctionalInterface
interface MyFunction {
    int apply(int a);
}
```

---

367. What is a Single Abstract Method (SAM) interface in Java 8?

- Functional Interface with **one abstract method**.
- Example: Runnable, Callable, Comparator

---

368. How to define a Functional Interface in Java 8?

Use @FunctionalInterface annotation:

```
@FunctionalInterface
interface MyFunction {
```

```
    int apply(int a);
}
```

---

369. Why do we need Functional Interface in Java?

✅ Supports Lambda expressions
✅ Enables functional programming
✅ Reduces boilerplate code

---

370. Is @FunctionalInterface mandatory for defining a Functional Interface?

- **No** – But using it ensures compile-time checking.

---

371. Differences between Collection and Stream API in Java 8

Aspect	Collection	Stream
Storage	Stores data	Does not store data
Processing	External iteration	Internal iteration
Execution	Sequential	Sequential or parallel

---

372. Main uses of Stream API in Java 8

✅ Filtering
✅ Mapping
✅ Reducing
✅ Sorting

Example:

```
List<Integer> evenNumbers = list.stream()
                            .filter(n -> n % 2 == 0)
                            .collect(Collectors.toList());
```

---

373. Differences between Intermediate and Terminal Operations in Streams

Operation	Type	Example
Intermediate	Returns stream	filter(), map()
Terminal	Ends stream	collect(), forEach()

---

374. What is a Spliterator in Java 8?

- Used for parallel processing in streams.
- Can split elements for parallel processing.

---

375. Differences between Iterator and Spliterator in Java 8

Feature	Iterator	Spliterator
Parallel	No	Yes
Characteristics	No	Yes
Splitting	No	Yes

---

376. What is Type Inference in Java 8?

- Compiler determines the data type automatically.

Example:

```
var x = 10; // Compiler infers int type
```

---

377. Does Java 7 support Type Inference?

✅ Only for diamond operator (<>)

Example:

```
List<String> list = new ArrayList<>();
```

---

378. How does Internal Iteration work in Java 8?

- Streams use internal iteration → **No need for loop**

Example:

```
list.stream().forEach(System.out::println);
```

---

379. Differences between Internal and External Iteration

Type	Example	Performance
Internal	stream.forEach()	Better
External	for loop	Slower

---

380. Advantages of Internal Iterator over External Iterator

✅ Better performance

✅ Simpler code

✅ Supports parallel processing

Example:

```
list.stream().forEach(System.out::println);
```

- **Advanced Java 8 Features and Concepts**

381. Applications where we should use Internal Iteration

✅ Parallel processing of large datasets

✅ Functional-style programming

✅ Data manipulation using Stream API

✅ Simplifying complex data processing

Example:

```
list.stream()
   .filter(n -> n % 2 == 0)
   .forEach(System.out::println);
```

---

382. Main Disadvantage of Internal Iteration over External Iteration

❌ **Less control** over the iteration process

❌ Cannot modify the collection during iteration

❌ Difficult to implement custom traversal logic

Example:

- Internal iteration example (less control):

```
list.stream().forEach(System.out::println);
```

- External iteration example (more control):

```
for (Integer i : list) {
   if (i > 5) break;
}
```

---

383. Can we provide implementation of a method in a Java Interface?

✅ **Yes** – Since **Java 8**, using **default** and **static** methods.

---

384. What is a Default Method in an Interface?

- A method in an interface with a **default implementation**.
- Allows adding new methods without breaking existing implementations.

Example:

```
interface MyInterface {
    default void show() {
        System.out.println("Default method");
    }
}
```

---

385. Why do we need Default Method in a Java 8 Interface?

✅ To add new functionality to interfaces without breaking backward compatibility.

✅ Provides a way to implement shared behavior.

Example:

```
interface MyInterface {
    default void show() {
        System.out.println("New feature");
    }
}
```

---

386. Purpose of Static Method in an Interface in Java 8

✅ Provides utility methods related to the interface

✅ No need to create an instance

Example:

```
interface MyInterface {
    static void print() {
        System.out.println("Static method");
    }
}
```

---

387. Core Ideas Behind the Date/Time API of Java 8

✅ **Immutability** – Thread-safe

✅ Based on **ISO-8601 standard**

✅ Clear distinction between **date** and **time**

✅ Supports **time zones**

---

388. Advantages of New Date/Time API over Old Date API

✅ Immutable and thread-safe

✅ Clear and consistent methods

✅ Improved formatting and parsing

---

389. Differences Between Legacy Date API and Java 8 Date/Time API

Feature	Legacy Date API	Java 8 Date/Time API
Thread Safety	Not thread-safe	Thread-safe
Package	java.util.Date	java.time.*
Mutability	Mutable	Immutable

Feature	Legacy Date API	Java 8 Date/Time API
Formatting	SimpleDateFormat	DateTimeFormatter

---

390. How to Get Duration Between Two Dates or Times in Java 8

Use Duration or Period:

```
LocalDate start = LocalDate.of(2023, 1, 1);
LocalDate end = LocalDate.of(2025, 1, 1);
Period period = Period.between(start, end);
System.out.println(period.getYears());
```

---

391. New Method Family for Processing Arrays on Multi-core Machines

✅ Arrays.parallelSort() – Uses multiple threads internally.

```
int[] arr = {5, 2, 8, 3};
Arrays.parallelSort(arr);
```

---

392. How Java 8 Solves Diamond Problem of Multiple Inheritance

✅ Compiler resolves conflict using the following rule:

1. **Class > Interface**
2. If multiple interfaces have same method, subclass must override it.

Example:

```
class MyClass implements InterfaceA, InterfaceB {
    @Override
    public void show() {
        InterfaceA.super.show(); // Resolve conflict
    }
}
```

---

393. Differences Between Predicate, Supplier, and Consumer

Interface	Description	Example
Predicate	Takes input, returns boolean	(x) -> x > 0
Supplier	Takes no input, returns value	() -> new Date()
Consumer	Takes input, returns nothing	x -> System.out.println(x)

---

394. Can We Define a Default Method Without default Keyword?

❌ **No** – Default method must be marked with default keyword.

---

395. Can a Class Implement Two Interfaces With Same Default Method?

✅ Yes – Must override the conflicting method.

Example:

```
interface A { default void show() {} }
interface B { default void show() {} }

class MyClass implements A, B {
    @Override
    public void show() {
        A.super.show(); // Resolving conflict
    }
}
```

---

396. How Java 8 Supports Multiple Inheritance

✅ Through **default methods** in interfaces

---

397. If Base Class and Interface Have Same Default Method, Which is Picked?

✅ Base class method is picked over interface method.

Example:

```
class Base {
   void show() { System.out.println("Base"); }
}

interface MyInterface {
   default void show() { System.out.println("Interface"); }
}

class MyClass extends Base implements MyInterface { }
```

➡️ Output: Base

---

398. What if Class, Parent Class, and Interface Have Same Method?

✅ Class definition is used.

---

399. Can We Access a Static Method of an Interface Using a Reference?

❌ No – Must call it using interface name.

Example:

```
MyInterface.print(); // Correct
```

---

400. How to Get Parameter Names Using Reflection in Java

Use Method.getParameters() from java.lang.reflect:

```
Method method = MyClass.class.getMethod("myMethod", String.class);
Parameter[] params = method.getParameters();
for (Parameter param : params) {
   System.out.println(param.getName());
}
```

---

401. What is Optional in Java 8?

✅ A container class to handle **null values** safely.

Example:

```
Optional<String> value = Optional.ofNullable(null);
```

---

402. Uses of Optional

✅ Avoid NullPointerException

✅ Provide default values

✅ Handle absence of value

---

403. Method in Optional That Provides Fallback Mechanism

✅ orElse()

Example:

```
String value = optional.orElse("Default");
```

---

404. How to Get Current Time Using Date/Time API

✅ LocalTime.now()

```
LocalTime now = LocalTime.now();
System.out.println(now);
```

---

405. Can We Define a Static Method in an Interface?

✅ Yes

Example:

```
interface MyInterface {
   static void print() {
      System.out.println("Static method");
   }
}
```

---

406. How to Analyze Dependencies in Java Classes/Packages

✅ Use **jdeps** tool:

```
jdeps MyClass.class
```

---

407. New JVM Arguments Introduced in Java 8

✅ -XX:+UseG1GC – Enable G1 Garbage Collector

✅ -XX:+PrintGCDetails – Print GC logs

---

408. Popular Annotations Introduced in Java 8

✅ @FunctionalInterface

✅ @Repeatable

✅ @SafeVarargs

---

409. What is StringJoiner in Java 8?

✅ Joins strings with a delimiter.

Example:

```
StringJoiner joiner = new StringJoiner(", ");
joiner.add("Apple").add("Banana");
System.out.println(joiner);
```

---

410. Type of a Lambda Expression

✅ Type of a lambda expression = **Functional Interface**

Example:

```
BiFunction<Integer, Integer, Integer> add = (a, b) -> a + b;
```

- **Java 8 Stream API and Lambda Expressions**

411. What is the Target Type of a Lambda Expression?

✅ The target type of a lambda expression is a **functional interface**.

✅ A lambda expression is assigned to a reference of a functional interface.

Example:

```
Runnable r = () -> System.out.println("Running");
```

Here, Runnable is the **target type**.

---

412. Differences Between an Interface with Default Method and an Abstract Class

Feature	Interface with Default Method	Abstract Class
Implementation	Can have default methods	Can have concrete and abstract methods
Multiple Inheritance	Can implement multiple interfaces	Can extend only one abstract class
Constructors	No constructors	Can have constructors
Accessibility	Cannot define instance fields	Can define instance fields

Example:

- Interface with default method:

```
interface MyInterface {
   default void show() {
      System.out.println("Default Method");
   }
}
```

- Abstract class:

```
abstract class MyClass {
   abstract void show();
}
```

---

413. What is the Stream API in Java?

✅ Introduced in **Java 8** to process collections in a **functional style**.

✅ Allows processing data in a **declarative manner** using pipeline operations.

Example:

```
List<Integer> list = Arrays.asList(1, 2, 3, 4);
list.stream().filter(n -> n % 2 == 0).forEach(System.out::println);
```

---

414. Main Advantages of Using the Stream API

✅ **Declarative code** – Less boilerplate code

✅ **Parallel processing** – Improves performance

✅ **Lazy evaluation** – Optimized execution

✅ **Functional style** – More readable code

---

415. How Do You Create a Stream?

✅ From a collection:

```
Stream<Integer> stream = list.stream();
```

✅ From an array:

```
Stream<String> stream = Stream.of("A", "B", "C");
```

✅ From generate or iterate:

```
Stream<Double> stream = Stream.generate(Math::random).limit(5);
```

---

416. What Are Intermediate and Terminal Operations?

Operation Type	Description	Example
Intermediate	Returns a stream (lazy)	map(), filter()
Terminal	Returns a non-stream value or action	forEach(), collect()

Example:

```
list.stream()
   .filter(n -> n > 2) // Intermediate
```

```
    .forEach(System.out::println); // Terminal
```

---

417. What is the map() Operation?

✅ Transforms each element in the stream.

```
list.stream()
    .map(n -> n * 2)
    .forEach(System.out::println);
```

---

418. How Does Filtering Work in Streams?

✅ filter() – Removes elements that don't match the predicate.

```
list.stream()
    .filter(n -> n > 2)
    .forEach(System.out::println);
```

---

419. Difference Between collect() and reduce()

Operation	Purpose	Example
collect()	Produces a collection	toList(), joining()
reduce()	Produces a single result	(a, b) -> a + b

Example of reduce():

```
int sum = list.stream().reduce(0, Integer::sum);
```

---

420. Can You Perform Parallel Processing with the Stream API?

✅ Yes, using parallelStream() or parallel().

```
list.parallelStream().forEach(System.out::println);
```

---

421. Purpose of flatMap() Operation

✅ Flattens a nested stream structure into a single stream.

```
List<List<Integer>> nestedList = Arrays.asList(Arrays.asList(1, 2), Arrays.asList(3, 4));
nestedList.stream()
    .flatMap(Collection::stream)
    .forEach(System.out::println);
```

---

422. How to Limit the Number of Elements in a Stream?

✅ limit() – Returns a stream of specified size.

```
list.stream().limit(2).forEach(System.out::println);
```

---

423. What Does skip() Do in a Stream?

✅ Skips the first **n** elements.

```
list.stream().skip(2).forEach(System.out::println);
```

---

424. What Are Primitive Streams?

✅ Specialized streams for primitive types:

- IntStream, LongStream, DoubleStream

Example:

```
IntStream.range(1, 5).forEach(System.out::println);
```

---

425. Difference Between findFirst() and findAny()

Method	Description

Method	Description
findFirst()	Returns the first element
findAny()	Returns any element (useful for parallel streams)

Example:

```
Optional<Integer> first = list.stream().findFirst();
```

---

426. Purpose of peek() Operation

✅ Used for debugging – performs an action without modifying the stream.

```
list.stream()
   .peek(System.out::println)
   .collect(Collectors.toList());
```

---

427. How to Combine Multiple Streams?

✅ Stream.concat()

```
Stream<Integer> stream1 = Stream.of(1, 2);
Stream<Integer> stream2 = Stream.of(3, 4);
Stream.concat(stream1, stream2).forEach(System.out::println);
```

---

428. allMatch(), anyMatch(), and noneMatch()

Method	Description
allMatch()	Returns true if all elements match the predicate
anyMatch()	Returns true if any element matches the predicate
noneMatch()	Returns true if no element matches the predicate

Example:

```
boolean allEven = list.stream().allMatch(n -> n % 2 == 0);
```

---

429. What is the Collectors Class?

✅ Utility class for reducing and collecting stream elements into a collection.

---

430. How to Use toList()

✅ Collect elements into a list.

```
List<Integer> result = list.stream().collect(Collectors.toList());
```

---

431. How to Use toMap() to Create a Map from a Stream?

✅ Convert stream elements into a Map.

```
Map<Integer, String> map = list.stream()
   .collect(Collectors.toMap(n -> n, n -> "Value" + n));
```

---

432. What Does the joining() Collector Do?

✅ Joins elements into a single string.

```
String result = list.stream()
   .map(String::valueOf)
   .collect(Collectors.joining(", "));
```

---

433. How to Use groupingBy()?

✅ Groups elements into a Map based on a classifier.

```
Map<Boolean, List<Integer>> evenOdd = list.stream()
```

```
    .collect(Collectors.groupingBy(n -> n % 2 == 0));
```

---

434. What is partitioningBy()?

✅ Similar to groupingBy() but returns a Map<Boolean, List<T>>.

```
Map<Boolean, List<Integer>> partition = list.stream()
    .collect(Collectors.partitioningBy(n -> n % 2 == 0));
```

---

435. What is reducing()?

✅ Performs a reduction operation on stream elements.

```
int sum = list.stream().collect(Collectors.reducing(0, Integer::sum));
```
