Q1

Firstly, OOP allows us to create reusable classes by creating multiple instances of that object. That means that we can write code and use that in different parts of our projects, therefore, it saves us time and keeps our codebase easy to maintain and organized.

Secondly, OOP allows us to hide the details of how an object works from the rest of the code to enjoy the functionalities in other parts without worrying about how the work is done.

Thirdly, OOP supports the concept of polymorphism. That allows us to treat the object as the parent class instance. Thus, we can write methods that work with any subclasses of a certain class rather than writing a separate method for each subclass, reducing code duplication and improving flexibility.

Last but not least, OOP allows a subclass to inherit the properties and methods of its superclass, making it easier to reuse existing code without duplicating functionality.

Java, Ruby, and C# are fully OOP languages. In addition, many popular languages, such as C++, Python, and Javascript, support OOP features alongside other programming paradigms.

Q2

Both abstract classes and interfaces establish blueprints for other classes, ensuring consistent implementation of methods and behaviours. However, they have distinct characteristics and use cases.

Java does not support multiple class-level inheritance so every class can extend only one class. However, a class can implement multiple interfaces.

Abstract classes can have constructors, even though they can’t be instantiated directly. Interfaces cannot have constructors at all.

Abstract classes can have instance variables, both static and non-static, and can be of any access modifier. Interfaces can have public static final fields (constants).

Interfaces cannot have protected or package-private methods. Abstract classes’ methods can have any access modifier.

Abstract classes can maintain state through instance variables. Interfaces can’t have instance variables, so they are more about defining behavior without the state.

Abstract classes provide a common base implementation for subclasses, sharing code, and maintaining state. Interfaces are better for defining contracts that any class can implement.

Q3

equals(): To determine semantic/logical equality between objects, we need the equals() method. The default behavior of the equals() is checking reference equality. However, this implementation often doesn't reflect meaningful equality. For example, two person objects with the same name and age should be equal even if they are different objects. By overriding, equality constraints can be specified.

hashCode(): Generates an integer hash code for an object, used in hash based collection like hashmap, hashset, etc. By default, it returns a unique integer based on the object's memory address. If equals() is overridden but hashCode() isn't, equal objects may have different hashcodes. This breaks hash-based collections. Overriding hashCode to ensure equal objects produce the same hash code is the solution.

Q4

If a class implements two interfaces with a default method with the same signature, it creates ambiguity about which implementation the class should inherit.
For example, class D implements interfaces B and C. In this example, if D tries to call display(), which is also implemented as default in interfaces B and C, the compiler won't know whether to use the default implementation from B or C.
Providing its own implementation of the display method in D by overriding the display() method explicitly tells Java which behaviour to use. If you want to use the default implementation from a specific interface, you can use the InterfaceName.super.display() syntax within the class D.

Q5

We need garbage collectors to simplify memory management for developers and to prevent a host of common programming errors. By automatically managing memory allocation and deallocation, the garbage collector eliminates issues such as forgetting to free an object, which leads to memory leaks or attempting to access memory that has already been freed, resulting in dangling pointers or other unpredictable behaviors. It provides an abstraction layer that shields developers from the low-level details of memory management.

A garbage collector determines which objects are in memory and still in use by tracing the graph of other references starting from a set of root objects. These roots are typically variables currently on the call stack, CPU registers, and static fields of classes. In Java, roots include static fields and JNI references. Any object that cannot be reached through any chain of references from the roots is considered unreachable or “garbage” and reclaimed, meaning it is made available for future allocation.

Q6

The static keyword in Java is used for memory management and to associate members with a class rather than with specific instances of that class. When a member is declared static, only one copy of that member exists in memory, regardless of how many objects of the class are created. Static variables are used for properties that are shared by all objects of a class, and static methods are generally used for utility functions that don't depend on the state of a specific object. Static blocks are used for performing initialization tasks that need to be done only once when the class is loaded into memory, especially for complex initializations of static variables (database connection)

Q7
Immutability in Java signifies that the state of an object remains constant throughout its lifespan from the moment of its construction onwards. Once an immutable object is initialized, its internal data cannot be altered.
Where:

- Concurrency and thread safety environments.
- Caching
- Functional Programming
- Data Integrity
- Representing Values

How:

- Creating immutable classes by declaring the class as final (prevents subclassing)
- Declaring all fields as private and final. Private ensures encapsulation, final guarantees that final can be assigned only once (initialize all finals during constructor)
- Not providing any setter methods.
- Using immutable collections such as List.of(), Set.of(), Map.of(), Guava immutable collections.

Why:

- Immutable objects cannot modified, so they are thread-safe.
- Reduces bugs by preventing state changes after creation. Eliminates a common source of bugs related to unexpected modifications and side effects.
- Caching. Immutable objects provide stable state, so there is no need to worry about the object being modified elsewhere in the application. Also they are reliable as cache keys.
- Functional Programming. A pure function is one that, given the same input, will always produce the same output and has no side effects. Immutable data structures are essential for writing pure functions.

Q8
Aggregation and Composition are both based on the "has-a" relationship, indicating that an object of one class holds a reference to an object of another class.

Characteristics	Composition	Aggregation
Lifecycle Dependency	Dependent (Part dies with Whole)	Independent

Ownership	Exclusive	Shared or None
Object Creation	Typically within the Whole	Often outside the Whole
Object Destruction	Part is destroyed with Whole	Part can exist after Whole is destroyed
Relationship Strength	Strong	Weak
UML Notation	Filled Diamond on the Whole	Hollow Diamond on the Whole

Q9
Coupling refers to the degree of interdependence between different modules or components in a software system. Low coupling implies that modules are relatively independent and minimal knowledge of each other's internal details. This makes the system more flexible, easier to maintain, and less prone to ripple effects when changes are made to one module. Low coupling can be achieved by:
- Using well-defined interfaces
- Minimizing direct dependencies
- Using dependency injection

Cohesion refers to the degree to which the responsibilities and functionality within a module or class are related and focused. High cohesion indicatesthat a module or class has a clear and well-defined purpose, and its internal components work together to achieve that purpose. This leads to more readable, maintainable, and reusable code. High cohesion can be achieved by:
- Following the Single Responsibility Principle
- Grouping related methods and data
- Avoiding "God classes"

Q10
Stack allocation refers to the process of assigning memory for local variables and function calls in the call stack. It happens automatically when a function is called and is freed immediately when the function ends. Memory is allocated in a contiguous block. It is less costly and easy to implement.

Heap allocation is allocated dynamically during the program execution. Unlike stack memory, heap memory is not freed automatically when a function ends. Instead, it requires manual deallocation or a garbage collector to reclaim unused memory. All runtime classes are stored in the heap. Memory is allocated in any random order. It is more costly and hard to implement

The object reference is stored in stack memory, but the actual object is stored in heap memory.

Q11

Exceptions: An event that disrupts the normal flow of a program, typically caused by errors. Java handles exceptions using try-catch blocks, allowing graceful recovery or termination.

1.Checked Exceptions:
*Checked at compile-time: Must be either try-catch or throws (declared in the method signature)
Examples: IOException, SQLException, ClassNotFoundException

2.Runtime Exceptions
*Not checked at compile-time: Usually caused by programming errors.
Examples: NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException

3.Errors
*Severe, unrecoverable issues: Typically JVM/system-level failures.
Examples:OutOfMemoryError, StackOverflowError, VirtualMachineError.

Q12
Code humans can understand and evolve effortlessly

Q13
Subclass defines a static method with the same signature as a static method in its superclass. The subclass method hides the superclass method, and the version called depends on the reference type, not the object type.
Overriding: Runtime polymorphism (depends on object type)
Hiding: Compile-time resolution (depends on reference type).

Q14
Abstraction: Hides implementation details and exposes only essential features. Achieved by abstract classes and interfaces. Focus on design-type structure.
Polymorphism: Allows objects of different classes to be treated as objects of a common superclass/interface. It is achieved by method overriding and runtime binding. Focus on runtime behavior flexibility.