Object Oriented Concepts

1. What are object-oriented concepts? What is the difference between object-based, object-oriented, and fully object-oriented language? What are advantages of Object Oriented Programming? What is data security?

- **Object-OrientOriented Concepts (Pillars of OOP):**
    - **Abstraction:** Hiding complex implementation details and showing only essential features of an object.
    - **Encapsulation:** Bundling data (attributes) and methods (functions) that operate on the data within a single unit (class), and restricting direct access to some of the object's components.
    - **Inheritance:** A mechanism where a new class (subclass/derived class) derives properties and behavior from an existing class (superclass/base class). It promotes code reusability.
    - **Polymorphism:** The ability of an object to take on many forms. It allows objects of different classes to be treated as objects of a common type.
- **Difference between Object-based, Object-oriented, and Fully Object-oriented language:**
    - **Object-based Language:** Supports object features like encapsulation and identity, but *does not* necessarily support all four pillars of OOP, particularly inheritance and polymorphism. Examples: JavaScript (historically, before ES6 classes), Ada.
    - **Object-oriented Language:** Supports all four core OOP pillars: Abstraction, Encapsulation, Inheritance, and Polymorphism. Examples: Java, C++, C#.
    - **Fully Object-oriented Language:** A language where *everything* is treated as an object, including primitive data types (like integers, booleans, etc.). Examples: Smalltalk, Ruby. Java is *not* fully object-oriented because it has primitive data types.
- **Advantages of Object-Oriented Programming:**
    - **Modularity:** Objects create modular structures, making code easier to understand, manage, and debug.
    - **Reusability:** Inheritance allows reusing existing code, reducing development time and effort.
    - **Maintainability:** Encapsulation makes it easier to modify and maintain code without affecting other parts of the system.
    - **Flexibility & Extensibility:** Polymorphism allows adding new features or modifying existing ones without changing the core code.
    - **Security (Data Hiding):** Encapsulation helps protect data from unauthorized access by hiding internal implementation details.
    - **Problem Solving:** Mimics real-world entities, making it easier to model and solve complex problems.
- **Data Security:** In the context of OOP, data security primarily refers to **Data Hiding** or **Information Hiding**, which is achieved through **Encapsulation**. It means preventing direct access to an object's internal state (attributes) and exposing only necessary methods to interact with that data. This protects the data from accidental or malicious modification.

2. What is class and object? Give real-life example.

- **Class:** A blueprint or a template for creating objects. It defines the attributes (data) and methods (behavior) that all objects of that type will have. It's a logical entity.
    - **Real-life Example:** The **blueprint** of a "Car". It defines what a car *is* (has wheels, an engine, seats) and what it *can do* (drive, brake, accelerate).
- **Object:** An instance of a class. It's a concrete entity created based on the class blueprint, having its own unique state (values for attributes) and behavior.
    - **Real-life Example:** Your specific **Toyota Camry** parked in your driveway. It's a real car (an instance of the "Car" blueprint) with its own color, mileage, and current speed. Another example would be a "Ferrari" object, also an instance of the "Car" class, but with different attributes.

3. What are characteristics of object? Explain them.

The three main characteristics of an object are:

- **State (Attributes/Properties):**
    - **Explanation:** The data or values stored within an object at a particular moment. It defines the object's current condition or characteristics.
    - **Example:** For a "Car" object, its state could include `color` (red), `make` (Honda), `model` (Civic), `currentSpeed` (60 mph), `mileage` (50,000 miles).
- **Behavior (Methods/Functions):**
    - **Explanation:** The actions or operations that an object can perform or that can be performed on it. These are typically defined by the methods of the class.
    - **Example:** For a "Car" object, its behavior could include `startEngine()`, `accelerate()`, `brake()`, `turn(direction)`, `honkHorn()`.
- **Identity:**
    - **Explanation:** The unique characteristic that distinguishes one object from another, even if they have the exact same state. Each object has a distinct identity in memory.
    - **Example:** Two "Car" objects might both be red Honda Civics with 50,000 miles, but they are still two separate, distinct cars (objects) with their own unique memory addresses and existence.

4. What is the need of getter and setter functions in class?

Getter and setter functions (also known as accessors and mutators) are crucial for implementing **encapsulation** and **data hiding**.

- **Need for Getters (Accessors):**
    - **Controlled Access:** They provide a standardized and controlled way to read the values of an object's private attributes.
    - **Validation (Implicit):** While getters usually just return values, they could potentially add logic like formatting or deriving a value before returning it.
    - **Flexibility for Future Changes:** If the internal representation of an attribute changes, only the getter method needs to be updated, not every piece of client code that accesses it.

- **Need for Setters (Mutators):**
    - **Controlled Modification:** They provide a standardized and controlled way to modify the values of an object's private attributes.
    - **Data Validation:** This is a primary reason. Setters can implement validation logic to ensure that an attribute is set only to a valid value (e.g., `setAge(age)` could check `if (age > 0 && age < 120)`). This prevents the object from entering an invalid state.
    - **Business Logic:** Setters can trigger other actions or update related attributes when a value changes.
    - **Encapsulation Enforcement:** By making attributes private and only allowing access through setters, you prevent direct, uncontrolled modification from outside the class.

5. What is abstraction and encapsulation. Give real-life example.

- **Abstraction:**
    - **Definition:** Abstraction is the process of simplifying complex reality by modeling classes based on essential properties and behaviors, while hiding unnecessary details. It focuses on *what* an object does rather than *how* it does it.
    - **Real-life Example:** When you drive a **car**, you interact with its essential features: steering wheel, accelerator, brake pedal. You don't need to know the intricate details of how the engine converts fuel into motion, how the brake fluid operates, or the complex wiring behind the dashboard. The car *abstracts away* its internal mechanics, providing a simplified interface for you to operate it.
- **Encapsulation:**
    - **Definition:** Encapsulation is the bundling of data (attributes) and the methods (functions) that operate on that data into a single unit (a class). It also involves restricting direct access to some of an object's components, usually by making attributes private and providing public getter/setter methods for controlled access. It focuses on *how* the data is protected and managed.
    - **Real-life Example:** Consider a **television**. The internal components (circuits, wires, speakers, screen components) are all enclosed within the TV's casing. You interact with the TV using a remote control or buttons on the outside (the public interface like `turnOn()`, `changeChannel()`, `adjustVolume()`). You cannot directly access or tamper with the internal wiring or components. This "packaging" and "hiding" of internal details within the casing is encapsulation.

Relationship: Abstraction and Encapsulation are complementary. Abstraction helps in designing the "what" (interface), while encapsulation helps in implementing the "how" (hidden internal details and controlled access).

6. What is polymorphism? What are its types? Explain them with examples.

- **Polymorphism:** The word "polymorphism" means "many forms." In OOP, it refers to the ability of an object to take on many forms or the ability of a method to perform different actions based on the object it is called on. It allows objects of different classes to be treated as objects of a common supertype.
- **Types of Polymorphism:**

1. **Compile-time Polymorphism (Static Polymorphism / Method Overloading):**
    - **Explanation:** Achieved through **method overloading**, where multiple methods in the same class have the same name but different parameter lists (different number of parameters, different types of parameters, or different order of parameters). The compiler decides which method to call at compile time based on the arguments provided.
    - **Example (Java):**

```
class Calculator {
    // Method to add two integers
    public int add(int a, int b) {
        return a + b;
    }

    // Overloaded method to add three integers
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // Overloaded method to add two doubles
    public double add(double a, double b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(5, 10));        // Calls int
add(int, int)
        System.out.println(calc.add(5, 10, 15));    // Calls int
add(int, int, int)
        System.out.println(calc.add(5.5, 10.5));    // Calls
double add(double, double)
    }
}
```

2. **Run-time Polymorphism (Dynamic Polymorphism / Method Overriding):**
    - **Explanation:** Achieved through **method overriding**, where a subclass provides a specific implementation for a method that is already defined in its superclass. The decision of which method to call is made at runtime based on the actual type of the object (not the reference type). This typically involves inheritance and `super` keyword (though not always directly used in the call).
    - **Example (Java):**

```
class Animal {
    public void makeSound() {
```

```
        System.out.println("Animal makes a sound");
    }
}

class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Dog barks");
    }
}

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Cat meows");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myDog = new Dog(); // Upcasting: Animal reference
to Dog object
        Animal myCat = new Cat(); // Animal reference to Cat
object
        Animal genericAnimal = new Animal();

        myDog.makeSound();         // Output: Dog barks (Runtime
decision)
        myCat.makeSound();         // Output: Cat meows (Runtime
decision)
        genericAnimal.makeSound(); // Output: Animal makes a sound
    }
}
```

7. What is method overloading? Which are the rules of method overloading? Why return type is not considered in method overloading?

- **Method Overloading:** It is a feature in OOP that allows a class to have multiple methods with the same name, but with different parameters (i.e., different method signatures). This allows a single method name to perform different actions based on the arguments passed to it. It's a form of **Compile-time Polymorphism**.
- **Rules of Method Overloading:**
    1. **Same Method Name:** All overloaded methods must have the exact same name.
    2. **Different Parameter List (Signature):** The parameter list must differ in at least one of these ways:
        - **Number of arguments:** e.g., `add(int a, int b)` vs `add(int a, int b, int c)`.
        - **Data type of arguments:** e.g., `add(int a, int b)` vs `add(double a, double b)`.
        - **Order of arguments (if types differ):** e.g., `print(int a, String s)` vs `print(String s, int a)`.

3. **Return Type (Optional):** The return type *can* be the same or different. It does not play a role in method overloading.
4. **Access Modifiers & Exception Handling (Optional):** Access modifiers (public, private, etc.) and checked exceptions *can* be different for overloaded methods, but they don't affect whether a method is overloaded or not.

- **Why return type is not considered in method overloading?** The compiler differentiates overloaded methods purely based on their **method signature**, which consists of the method name and the parameter list. The return type is *not* part of the method signature. If only the return type were different, the compiler wouldn't be able to determine which method to invoke when the method is called, as the call itself does not specify a return type. For example:

```
class Example {
    public int getValue() { return 1; }
    public String getValue() { return "one"; } // This would be an error!
}
// If you call: Example obj = new Example(); obj.getValue();
// The compiler wouldn't know if you want the int or String version.
```

To avoid such ambiguity, the return type is explicitly excluded from the method overloading rules.

8. What are different types of hierarchy? When to use which one?

In OOP, "hierarchy" generally refers to the structural relationships between classes. The two main types are:

1. **Inheritance Hierarchy (IS-A Relationship):**
    - **Explanation:** This hierarchy represents a "is a type of" relationship, where a subclass inherits properties and behaviors from its superclass. It's used for **specialization** and **code reuse**. It forms a tree-like structure.
    - **Example:** `Dog` **is a** `Animal`, `Car` **is a** `Vehicle`.
    - **When to use:**
        - When you have a strong "is a" relationship between entities (e.g., a specific type is a more general type).
        - To promote code reuse and reduce redundancy (common behavior in base class, specific behavior in derived classes).
        - When you want to model a generalization-specialization relationship.
        - When you need to achieve run-time polymorphism (method overriding).
2. **Composition/Aggregation Hierarchy (HAS-A Relationship):**
    - **Explanation:** This hierarchy represents a "has a" or "part-of" relationship, where one class contains an object of another class as an attribute. It's used for **building complex objects from simpler ones** and promotes **modularity** and **flexibility**. This can be further divided into:
        - **Composition (Strong HAS-A):** The contained object cannot exist independently of the container object. If the container is destroyed, the contained object is also destroyed. (e.g., `House` **has a** `Room` - a room cannot exist without a house).

        - **Aggregation (Weak HAS-A):** The contained object can exist independently of the container object. (e.g., `Department` **has** `Professors` - professors can exist without a department).
    - **Example:** `Car` **has an** `Engine`, `Student` **has an** `Address`.
    - **When to use:**
        - When you have a "has a" relationship between objects.
        - To create complex objects by combining simpler, independent objects.
        - To promote loose coupling and high cohesion (changes in one part don't necessarily affect others).
        - When you want to reuse components by assembling them into larger structures.
        - It is generally preferred over inheritance when there isn't a clear "is-a" relationship, as it offers more flexibility and avoids issues like the "fragile base class" problem associated with deep inheritance hierarchies.

9. What is the difference between method overloading and method overriding?

Feature	Method Overloading	Method Overriding
Concept	Multiple methods with same name, different signatures.	Subclass provides specific implementation for a superclass method.
Relationship	Within the **same class** (or same class hierarchy, but within the current class's scope).	Between a **superclass and a subclass** (requires inheritance).
Signature	**Must be different** (number, type, or order of parameters).	**Must be the same** (same name, same parameters, same return type or covariant return type).
Return Type	Can be same or different. It is **not considered** for differentiation.	Must be the same or a covariant return type (subclass of original).
Access Modifier	Can be different.	Cannot be *more restrictive* than the overridden method.
`static` **methods**	Can be overloaded.	Cannot be overridden (can be "hidden" or "shadowed").
`final` **methods**	Can be overloaded.	Cannot be overridden.
`private` **methods**	Can be overloaded.	Cannot be overridden (not visible to subclass).
Polymorphism Type	**Compile-time polymorphism** (Static binding).	**Run-time polymorphism** (Dynamic binding).
Decision	Decided by the **compiler** at compile time.	Decided by the **JVM** at runtime.
Example	`add(int, int)` vs `add(double, double)`	`Animal.makeSound()` overridden by `Dog.makeSound()`

10. What is object slicing? Explain object slicing in context of up-casting?

- **Object Slicing:** Object slicing occurs when an object of a derived class is assigned to an object of its base class by **value**. In this process, the "extra" data members (attributes and methods) specific to the derived class are "sliced off" or truncated, and only the base class portion of the object is copied. The original derived class object's specialized behavior and data are lost.
- **Object Slicing in Context of Up-casting:** Up-casting itself (assigning a derived class object to a base class *reference* or *pointer*) does *not* cause object slicing. For example, in Java:

```
// Java Example (no slicing with references)
class Base { int b; }
class Derived extends Base { int d; }

public class Main {
    public static void main(String[] args) {
        Derived derivedObj = new Derived();
        derivedObj.b = 1;
        derivedObj.d = 2;

        Base baseRef = derivedObj; // Up-casting (by reference)
        // baseRef points to the *entire* derivedObj.
        // No slicing occurs. derivedObj.d is still there, just not
accessible via baseRef.
        // If you later downcast, you can retrieve it.
        System.out.println(baseRef.b); // Output: 1
        // System.out.println(baseRef.d); // Compile-time error: d is not in
Base
    }
}
```

Object slicing primarily occurs when you perform an up-cast **by value** (e.g., C++ copy constructor/assignment, or passing by value). In Java, objects are always passed by reference, so "slicing" in the C++ sense where part of the object is literally destroyed in memory doesn't happen with object assignments. However, the *effect* of losing derived class specific information is similar if you were to assign a derived object to a *new* base type variable:

```
// Java equivalent of "slicing" the effect - creating a new Base object from
Derived
class Base {
    int b;
    public Base(int b) { this.b = b; }
    public void display() { System.out.println("Base display: " + b); }
}

class Derived extends Base {
    int d;
    public Derived(int b, int d) {
        super(b);
```

```
        this.d = d;
    }
    @Override
    public void display() { System.out.println("Derived display: " + b + ",
" + d); }
    public void derivedOnlyMethod() { System.out.println("Derived specific
method."); }
}

public class Main {
    public static void main(String[] args) {
        Derived derivedObject = new Derived(10, 20);
        derivedObject.display();          // Output: Derived display: 10, 20
        derivedObject.derivedOnlyMethod(); // Output: Derived specific
method.

        // This is NOT upcasting by reference. This is creating a new Base
object
        // and initializing its Base part from the derivedObject's Base
part.
        // This is the closest Java equivalent to the *concept* of slicing.
        Base baseObject = new Base(derivedObject.b); // Or if Base had a
copy constructor
        // The 'd' member of Derived is not copied to baseObject.
        // baseObject only contains the 'b' part.
        baseObject.display();             // Output: Base display: 10
        // baseObject.derivedOnlyMethod(); // Compile-time error
    }
}
```

In C++, where objects can be assigned by value, explicit slicing occurs:

```
// C++ Example demonstrating slicing
class Base {
public:
    int b;
    void display() { std::cout << "Base display: " << b << std::endl; }
};

class Derived : public Base {
public:
    int d;
    void display() { std::cout << "Derived display: " << b << ", " << d <<
std::endl; }
};

int main() {
    Derived d_obj;
    d_obj.b = 10;
    d_obj.d = 20;
    d_obj.display(); // Output: Derived display: 10, 20
```

```
    Base b_obj = d_obj; // Object Slicing! Only the Base part is copied.
    b_obj.display();     // Output: Base display: 10 (Derived's 'd' is lost,
Derived's method is not called)
    // b_obj.d; // Error: Base has no member 'd'

    // To avoid slicing, use pointers or references for polymorphism:
    Base* b_ptr = &d_obj; // Upcasting with a pointer
    b_ptr->display();     // Output: Derived display: 10, 20 (if display is
virtual)
    return 0;
}
```

The main takeaway is that slicing happens when a derived object is *assigned by value* to a base type variable, effectively chopping off the derived-specific parts. It's avoided by using pointers or references when working with polymorphic types.

11. What is down-casting and when it is required? Explain with code.

- **Down-casting:** Down-casting is the process of converting a reference to a superclass type into a reference to a subclass type. It's the opposite of up-casting. It's typically performed using an explicit type cast.
- **When it is required:** Down-casting is required when you have an object that is currently referenced by its superclass type, but you need to access specific methods or attributes that are unique to its actual subclass type. This often happens after up-casting has occurred.

  Example Scenario: Imagine a list of `Animal` objects. You iterate through them, and for some `Animal`s that you know are specifically `Dog`s, you want to call a `bark()` method that only `Dog`s have (and not all `Animal`s).

- **Explanation with Code (Java):**

```
class Animal {
    public void eat() {
        System.out.println("Animal is eating.");
    }
}

class Dog extends Animal {
    public void eat() { // Overriding base method
        System.out.println("Dog is eating kibble.");
    }
    public void bark() { // Dog-specific method
        System.out.println("Woof woof!");
    }
}

class Cat extends Animal {
    public void eat() { // Overriding base method
        System.out.println("Cat is eating fish.");
    }
```

```
    public void meow() { // Cat-specific method
        System.out.println("Meow!");
    }
}

public class DowncastingExample {
    public static void main(String[] args) {
        // Up-casting: Dog object is referred to by an Animal reference
        Animal myAnimal = new Dog(); // This is valid upcasting
        myAnimal.eat(); // Calls Dog's eat() due to polymorphism

        // myAnimal.bark(); // Compile-time error! 'bark()' is not defined
in Animal.
                            // We need to downcast to access Dog-specific
methods.

        // Down-casting: Casting Animal reference back to Dog reference
        if (myAnimal instanceof Dog) { // Safely check type before casting
            Dog myDog = (Dog) myAnimal; // Explicit down-cast
            myDog.bark(); // Now we can call Dog's specific method
            myDog.eat();  // Still calls Dog's eat()
        } else {
            System.out.println("myAnimal is not a Dog.");
        }

        // Example of unsafe downcasting (leads to ClassCastException)
        Animal anotherAnimal = new Cat();
        anotherAnimal.eat(); // Calls Cat's eat()

        // Try to downcast a Cat object to Dog:
        try {
            Dog newDog = (Dog) anotherAnimal; // This will throw
ClassCastException at runtime
            newDog.bark();
        } catch (ClassCastException e) {
            System.out.println("Error: Cannot cast Cat to Dog. " +
e.getMessage());
        }
    }
}
```

Key Points:

- Down-casting is inherently risky because the runtime type of the object might not actually be the target subclass.
- Always use `instanceof` (in Java) or `dynamic_cast` (in C++) to check the actual type of the object before performing a down-cast to avoid `ClassCastException` (Java) or runtime errors (C++).
- If you find yourself frequently using down-casting, it might indicate a potential flaw in your class design. Often, polymorphism through method overriding is a cleaner alternative.

12. What do you know about association, composition and aggregation? Explain with the help of example.

These terms describe different types of relationships between classes, forming part of the "has-a" hierarchy (as opposed to "is-a" for inheritance).

1. **Association:**
    - **Definition:** A broad term that represents a general relationship between two or more independent classes. It typically describes that one object "uses" or "knows about" another. There's no ownership, and both objects can exist independently. It's the weakest form of relationship.
    - **Keyword:** "uses a" or "knows about"
    - **Characteristics:**
        - Loosely coupled.
        - Objects can exist independently.
        - Represented by a simple line connecting classes in UML.
    - **Example:** A `Student` and a `Course`. A student can exist without being enrolled in a course, and a course can exist without any students enrolled. A student "knows about" the courses they are taking, and a course "knows about" the students enrolled.

```
class Student {
    String name;
    // List of courses this student is enrolled in
    // Does not "own" the courses, just references them
    List<Course> courses;
    // ...
}

class Course {
    String title;
    // List of students enrolled in this course
    List<Student> students;
    // ...
}
```

2. **Aggregation:**
    - **Definition:** A specialized form of association that represents a "part-of" or "has-a" relationship where the "part" can exist independently of the "whole." It implies a weaker form of ownership. If the "whole" object is destroyed, the "part" objects are not necessarily destroyed.
    - **Keyword:** "has a" (weak ownership)
    - **Characteristics:**
        - "Part" can exist without the "whole".

- "Whole" can exist without the "part".
- Represented by a hollow diamond on the "whole" side of the relationship in UML.

- **Example:** A `Department` and a `Professor`. A department "has" professors. However, a professor can exist and teach (perhaps in another department or independently) even if a specific department ceases to exist.

```
class Professor {
    String name;
    String specialization;
    // ...
}

class Department {
    String name;
    List<Professor> professors; // Department "has" professors

    public Department(String name) {
        this.name = name;
        this.professors = new ArrayList<>();
    }

    public void addProfessor(Professor prof) {
        this.professors.add(prof);
    }
    // ...
}

// Usage:
// Professor prof1 = new Professor("Dr. Smith");
// Department dept1 = new Department("Computer Science");
// dept1.addProfessor(prof1);
// If dept1 is removed, prof1 still exists independently.
```

3. **Composition:**

- **Definition:** A stronger form of aggregation that also represents a "part-of" or "has-a" relationship, but where the "part" cannot exist independently of the "whole." It implies a strong ownership. If the "whole" object is destroyed, the "part" objects are also destroyed or become meaningless. The lifecycle of the "part" is dependent on the "whole."

- **Keyword:** "has a" (strong ownership, "owns a")

- **Characteristics:**

  - "Part" cannot exist without the "whole".
  - If "whole" is destroyed, "part" is also destroyed.
  - Represented by a filled diamond on the "whole" side of the relationship in UML.

- **Example:** A `House` and a `Room`. A house "has" rooms. A room cannot logically exist on its own if the house is demolished. The room is an integral part of the house's existence.

```
class Room {
    String type; // e.g., "Bedroom", "Kitchen"
    double area;

    public Room(String type, double area) {
        this.type = type;
        this.area = area;
    }
    // ...
}

class House {
    String address;
    private List<Room> rooms; // House "owns" rooms

    public House(String address) {
        this.address = address;
        this.rooms = new ArrayList<>();
        // Rooms are often created along with the house
        this.rooms.add(new Room("Living Room", 300));
        this.rooms.add(new Room("Kitchen", 200));
    }

    public void addRoom(Room room) { // Can also add externally created
rooms
         this.rooms.add(room);
    }
    // ...
}

// Usage:
// House myHouse = new House("123 Main St");
// // The rooms within myHouse are created and managed by the House
object.
// // If myHouse object is garbage collected, its Room objects are also
eligible for GC.
```

Summary Table:

Relationship	Meaning	Independent Existence of Part	Ownership	UML Symbol	Example
Association	General connection	Yes	None	Simple line	Student & Course
Aggregation	Part-of (weak)	Yes	Weak	Hollow diamond	Department & Professor
Composition	Part-of (strong, integral)	No	Strong	Filled diamond	House & Room

13. What are different types of inheritance? Explain with the help of example. What are problems with multiple inheritance?

Inheritance is a fundamental OOP concept where a new class (derived class or subclass) is created from an existing class (base class or superclass). It promotes code reusability and establishes an "is-a" relationship.

Types of Inheritance (Common in OOP, typically referring to class inheritance):

1. **Single Inheritance:**
    - **Explanation:** A class inherits from only one base class. This is the simplest and most common type.
    - **Example (Java):**

```
class Animal { // Base class
    void eat() { System.out.println("Animal eats."); }
}

class Dog extends Animal { // Dog inherits from Animal
    void bark() { System.out.println("Dog barks."); }
}
```

2. **Multilevel Inheritance:**
    - **Explanation:** A class inherits from another class, which in turn inherits from another class. This forms a chain of inheritance.
    - **Example (Java):**

```
class Grandparent {
    void wealth() { System.out.println("Grandparent's wealth."); }
}

class Parent extends Grandparent { // Parent inherits from Grandparent
    void wisdom() { System.out.println("Parent's wisdom."); }
}

class Child extends Parent { // Child inherits from Parent (and
indirectly Grandparent)
    void youth() { System.out.println("Child's youth."); }
}
```

3. **Hierarchical Inheritance:**
    - **Explanation:** Multiple classes inherit from a single base class. It's like a family tree where one parent has multiple children.
    - **Example (Java):**

```
class Vehicle { // Base class
    void start() { System.out.println("Vehicle started."); }
}

class Car extends Vehicle { // Car inherits from Vehicle
    void drive() { System.out.println("Car is driving."); }
}

class Bicycle extends Vehicle { // Bicycle also inherits from Vehicle
    void pedal() { System.out.println("Bicycle is pedaling."); }
}
```

4. **Multiple Inheritance:**
    - **Explanation:** A class inherits from *multiple* base classes. (Note: Java does *not* support multiple inheritance of classes to avoid complexity, but achieves a similar concept through **interfaces**). C++ supports multiple inheritance for classes.
    - **Example (Conceptual, or C++):**

```
// Imagine in a language that supports multiple inheritance
class Flyer { void fly(); }
class Swimmer { void swim(); }
class Duck extends Flyer, Swimmer { // Duck inherits from both
    // ...
}
```

    - **Problems with Multiple Inheritance (The "Diamond Problem"):** The primary reason why languages like Java don't support multiple inheritance of classes is the "Diamond Problem" (or Deadly Diamond of Death).
        - **Scenario:** Consider a hierarchy where `Class D` inherits from `Class B` and `Class C`, and both `B` and `C` inherit from `Class A`. If `Class A` has a method (e.g., `doSomething()`) that `B` and `C` both override (or not), and `D` tries to call `doSomething()`, it becomes ambiguous which version of `doSomething()` `D` should inherit: the one from `B` or the one from `C`?
        - **Illustration:**

```
  A
 / \
B   C
 \ /
  D
```

        If `A` has `methodX()`, and `B` and `C` both inherit it. If `D` inherits from both `B` and `C`, and `D` calls `methodX()`, which `methodX()` does it get? The one from `B` or `C`? This ambiguity makes the class hierarchy complex and difficult to manage.
        - **Other Problems:**

- **Increased Complexity:** The class hierarchy becomes more intricate, making it harder to understand and maintain the code.
- **Constructor Chaining Issues:** Managing constructor calls from multiple parent classes can be complicated.
- **Reduced Readability and Debugging:** Tracing the origin of inherited methods and attributes becomes more challenging.

Java's Solution: Java avoids the Diamond Problem by disallowing multiple inheritance of classes. Instead, it uses **interfaces**, which allow a class to implement multiple contracts (behaviors) without inheriting implementation details, thus preventing the ambiguity. Java 8 introduced `default` methods in interfaces, which provide a limited form of implementation inheritance for interfaces, but with clear resolution rules to prevent ambiguity (e.g., explicit override required if multiple default methods conflict).

14. What is the difference between interface, abstract class and non-abstract class? Which one to use where?

Feature	Interface	Abstract Class	Non-Abstract (Concrete) Class
Purpose	Defines a **contract** for behavior. "What a class *can do*."	Provides a **partial implementation** and a common base for related classes. "What a class *is* (partially defined)."	Provides a **complete implementation** for creating objects. "What a class *is* (fully defined)."
Instantiation	Cannot be instantiated directly.	Cannot be instantiated directly.	Can be instantiated directly.
Abstract Methods	All methods implicitly `public abstract` (before Java 8). From Java 8, can have `default` and `static` methods.	Can have abstract (no body) and concrete (with body) methods.	All methods must have a body (unless it's an abstract class itself).
Concrete Methods	Can have `default` and `static` methods (Java 8+).	Can have concrete methods.	Can have concrete methods.
Variables	Only `public static final` constants (implicitly).	Can have `final`, `non-final`, `static`, and `non-static` variables.	Can have `final`, `non-final`, `static`, and `non-static` variables.
Access Modifiers	Methods and variables are implicitly `public`.	Can have any access modifier (public, protected, default, private) for methods and variables.	Can have any access modifier for methods and variables.
Inheritance	A class `implements` an interface. Can implement multiple interfaces.	A class `extends` an abstract class. Can extend only one abstract class.	A class `extends` another concrete/abstract class. Can extend only one.

Feature	Interface	Abstract Class	Non-Abstract (Concrete) Class
Constructors	Cannot have constructors.	Can have constructors. (Called by subclass constructor).	Can have constructors.
`main` **method**	Can have `main` method (from Java 8 for `static` methods).	Can have `main` method.	Can have `main` method.
Multiple Inheritance	Achieves multiple inheritance of **types/behavior**.	No multiple inheritance of classes.	No multiple inheritance of classes.

When to use which one:

1. **Interface:**
    - **Use when:** You want to define a contract for behavior that unrelated classes might share. It describes a set of actions that an object can perform, regardless of its underlying implementation.
    - **Examples:** `Runnable`, `Comparable`, `List`, `Serializable`.
    - **Scenario:** If you have `Car`, `Bicycle`, `Airplane` classes, and all of them can `start()`, `stop()`, and `move()`, you could define an `Movable` interface. Each class then provides its specific implementation of these methods. This is a "can do" relationship.
2. **Abstract Class:**
    - **Use when:** You have a strong "is-a" relationship and want to provide a common base class with some shared implementation, but you also need to enforce that certain methods must be implemented by subclasses. It's a partial blueprint.
    - **Examples:** `AbstractList`, `HttpServlet`.
    - **Scenario:** If you have `Dog`, `Cat`, `Bird` classes, they all `Animal`s. They might share common behaviors like `eat()` (which can have a default implementation) and `sleep()`, but each `Animal` type might have a unique `makeSound()` behavior that *must* be defined by the subclass. You can make `Animal` an abstract class with an abstract `makeSound()` method.
3. **Non-Abstract (Concrete) Class:**
    - **Use when:** You need to create a fully implemented, instantiable object. It provides a complete blueprint for a specific entity.
    - **Examples:** `String`, `ArrayList`, `HashMap`, `System`.
    - **Scenario:** When you need a specific, ready-to-use object like a `Customer`, `Product`, or `Order`. All methods have concrete implementations, and you can directly create instances of this class using `new`.

In short:

- **Interface:** Defines *what* a class *does* (contract/capability). "Can do this."
- **Abstract Class:** Defines *what* a class *is*, but with incomplete details that subclasses must fill in. "Is a (partially defined) type of this."

- **Concrete Class:** Defines *what* a class *is*, with full details and can be instantiated. "Is a fully defined type of this."

15. Which are the different types of design pattern? Explain singleton design pattern.

- **Design Patterns:** Reusable solutions to commonly occurring problems in software design. They are not ready-to-use code, but rather templates or blueprints that can be adapted to specific situations.
- **Different Types of Design Patterns (Categorized by GoF - Gang of Four):**
    1. **Creational Patterns:** Deal with object creation mechanisms, trying to create objects in a manner suitable for the situation.
        - Examples: **Singleton**, Factory Method, Abstract Factory, Builder, Prototype.
    2. **Structural Patterns:** Deal with the composition of classes and objects. They explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
        - Examples: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy.
    3. **Behavioral Patterns:** Deal with the communication between objects and how they interact to fulfill a complex task. They focus on how objects distribute responsibilities.
        - Examples: Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.
- **Singleton Design Pattern (Creational Pattern):**
    - **Definition:** The Singleton pattern ensures that a class has **only one instance** and provides a **global point of access** to that instance. It is used when you need exactly one object to coordinate actions across the system.
    - **Purpose/Problem It Solves:**
        - Ensures a class has only one instance.
        - Provides a global access point to that instance.
        - Controls instantiation from outside the class.
    - **Common Use Cases:**
        - Logging classes.
        - Configuration managers.
        - Database connection pools.
        - Thread pools.
        - Printer spoolers.
    - **Implementation Steps (Lazy Initialization - Thread-Safe):**
        1. **Private Constructor:** Make the constructor `private` to prevent direct instantiation from outside the class.

2. **Private Static Instance Variable:** Declare a `private static` variable of the same class type. This will hold the single instance.
3. **Public Static `getInstance()` Method:** Provide a `public static` method that returns the instance. This method checks if the instance already exists; if not, it creates it.

- **Example (Java - Thread-Safe using Double-Checked Locking):**

```
public class SingletonLogger {

    // 2. Private static instance variable (volatile for thread safety
in DCL)
    private static volatile SingletonLogger instance;

    // Optional: Instance-specific state
    private String logFile = "application.log";

    // 1. Private constructor to prevent direct instantiation
    private SingletonLogger() {
        // Prevent instantiation via reflection (optional but good
practice)
        if (instance != null) {
            throw new RuntimeException("Use getInstance() to get the
single instance.");
        }
        System.out.println("SingletonLogger instance created."); // For
demonstration
    }

    // 3. Public static method to provide global access
    public static SingletonLogger getInstance() {
        // First check (outside synchronized block) - avoids locking
overhead
        if (instance == null) {
            // Synchronized block to ensure thread safety during
instance creation
            synchronized (SingletonLogger.class) {
                // Second check (inside synchronized block) - protects
against multiple threads
                if (instance == null) {
                    instance = new SingletonLogger();
                }
            }
        }
        return instance;
    }

    // Business methods
    public void log(String message) {
        System.out.println("LOG to " + logFile + ": " + message);
    }

    // Example usage:
```

```
    public static void main(String[] args) {
        SingletonLogger logger1 = SingletonLogger.getInstance();
        logger1.log("Application started.");

        SingletonLogger logger2 = SingletonLogger.getInstance();
        logger2.log("User logged in.");

        // Verify that both references point to the same instance
        System.out.println("Are logger1 and logger2 the same instance?
" + (logger1 == logger2));
    }
}
```

In Java, the **Enum Singleton** is often considered the best approach for its inherent thread-safety and protection against serialization/reflection issues:

```
public enum EnumSingletonLogger {
    INSTANCE; // The single instance

    private String logFile = "application.log";

    public void log(String message) {
        System.out.println("ENUM LOG to " + logFile + ": " + message);
    }

    public static void main(String[] args) {
        EnumSingletonLogger logger1 = EnumSingletonLogger.INSTANCE;
        logger1.log("Application started (enum).");

        EnumSingletonLogger logger2 = EnumSingletonLogger.INSTANCE;
        logger2.log("User logged in (enum).");

        System.out.println("Are enum logger1 and logger2 the same
instance? " + (logger1 == logger2));
    }
}
```

Java Programming

1. What is a collection framework? Explain any 2 classes in Java Collections.

- **Java Collection Framework (JCF):** A unified architecture for representing and manipulating collections. It provides interfaces and classes to store and process groups of objects. JCF reduces programming effort, increases performance, and allows interoperability between unrelated APIs.
- **Key Interfaces:** `Collection`, `List`, `Set`, `Queue`, `Map` (Map is not strictly part of `Collection` but is part of the framework).
- **Explanation of 2 Classes:**

1. `ArrayList` **(Implements** `List` **interface):**

    - **Description:** A resizable array implementation of the `List` interface. It's essentially a dynamic array that grows its capacity as elements are added. It maintains insertion order and allows duplicate elements.
    - **Characteristics:**
        - **Indexed Access:** Elements can be accessed by their integer index (`get(index)`).
        - **Order:** Maintains insertion order.
        - **Duplicates:** Allows duplicate elements.
        - **Nulls:** Allows null elements.
        - **Performance:**
            - `get(index)` and `set(index)` are O(1) (constant time).
            - `add(element)` is amortized O(1).
            - `add(index, element)` and `remove(index)` are O(n) (linear time) because elements need to be shifted.
        - **Thread Safety:** Not thread-safe.

2. `HashMap` **(Implements** `Map` **interface):**

    - **Description:** A hash table-based implementation of the `Map` interface. It stores data in key-value pairs. It provides efficient storage and retrieval of elements based on a hash of the key.
    - **Characteristics:**
        - **Key-Value Pairs:** Stores data as unique keys mapped to values.
        - **Order:** Does not guarantee any specific order of elements. (Insertion order is not maintained).
        - **Keys:** Keys must be unique.
        - **Values:** Values can be duplicated.
        - **Nulls:** Allows one null key and multiple null values.
        - **Performance:**
            - `put()`, `get()`, `remove()` operations are typically O(1) on average, assuming good hash function distribution. In worst case (many collisions or poorly implemented `hashCode()`), it can degrade to O(n).
        - **Thread Safety:** Not thread-safe.

2. What is functional programming? Explain stream programming with examples.

- **Functional Programming (FP):** A programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. Key principles include:
    - **Pure Functions:** Functions that always produce the same output for the same input and have no side effects (don't modify external state).
    - **Immutability:** Data structures are not modified after creation; instead, new ones are created.
    - **First-Class Functions:** Functions can be treated like any other variable (assigned, passed as arguments, returned from other functions).
    - **Declarative Style:** Focuses on *what* to do rather than *how* to do it.
- **Stream Programming (Java 8+):**

- **Explanation:** Java's Stream API is a powerful feature introduced in Java 8 that supports functional-style operations on collections of objects. A stream is a sequence of elements from a source (like a `Collection`, an array, or an I/O channel) that supports aggregate operations (e.g., filter, map, reduce) without modifying the source data. Streams are declarative, lazy-evaluated, and can be processed sequentially or in parallel.
- **Core Concepts:**
  - **Source:** The collection or array from which the stream originates.
  - **Intermediate Operations:** Operations that transform a stream into another stream (e.g., `filter()`, `map()`, `sorted()`). They are lazy; they don't execute until a terminal operation is called.
  - **Terminal Operations:** Operations that produce a result or a side effect (e.g., `forEach()`, `collect()`, `reduce()`, `count()`). They trigger the execution of all intermediate operations.
- **Example:**

```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8,
9, 10);

        // --- Imperative Approach ---
        System.out.println("Imperative Approach:");
        List<Integer> evenSquaresImperative = new ArrayList<>();
        for (int number : numbers) {
            if (number % 2 == 0) { // Filter even numbers
                evenSquaresImperative.add(number * number); // Square
them
            }
        }
        System.out.println(evenSquaresImperative); // Output: [4, 16,
36, 64, 100]

        // --- Functional/Stream Approach ---
        System.out.println("\nFunctional/Stream Approach:");
        List<Integer> evenSquaresStream = numbers.stream() // Get a
stream from the list
            .filter(n -> n % 2 == 0)      // Intermediate operation:
filter out odd numbers
            .map(n -> n * n)              // Intermediate operation:
transform each even number to its square
            .collect(Collectors.toList());// Terminal operation:
collect the results into a new List

        System.out.println(evenSquaresStream); // Output: [4, 16, 36,
```

```
 64, 100]

        // Another example: Calculate sum of all odd numbers
        int sumOfOddNumbers = numbers.stream()
            .filter(n -> n % 2 != 0) // Filter odd numbers
            .reduce(0, Integer::sum); // Terminal operation: sum them
 up (0 is initial value)

        System.out.println("Sum of odd numbers: " + sumOfOddNumbers);
 // Output: 25 (1+3+5+7+9)
    }
 }
```

The Stream API makes code more concise, readable, and easier to parallelize, aligning with functional programming principles.

3. Explain multithreading using examples. Explain standard examples like a producer-consumer problem.

- **Multithreading:** In computer science, multithreading is the ability of a CPU or a single core in a multi-core processor to provide multiple threads of execution concurrently. In Java, multithreading allows a program to execute multiple parts (threads) concurrently within the same process. Each thread runs independently but shares the same memory space of the parent process.
- **Advantages:**
  - **Increased Responsiveness:** UI remains responsive while background tasks run.
  - **Better Resource Utilization:** CPU can be busy with one thread while another waits for I/O.
  - **Improved Performance:** Tasks can be broken down and run in parallel on multi-core processors.
  - **Simplified Design:** Complex tasks can be broken into simpler, concurrent sub-tasks.
- **Example (Simple Multithreading):**

```
class MyThread extends Thread {
    private String threadName;

    MyThread(String name) {
        threadName = name;
        System.out.println("Creating " +  threadName );
    }

    public void run() {
        System.out.println("Running " +  threadName );
        try {
            for(int i = 4; i > 0; i--) {
                System.out.println("Thread: " + threadName + ", " + i);
                // Let the thread sleep for a while.
                Thread.sleep(50);
            }
        } catch (InterruptedException e) {
            System.out.println("Thread " +  threadName + " interrupted.");
```

```
        }
        System.out.println("Thread " +  threadName + " exiting.");
    }
}

public class SimpleMultithreading {
    public static void main(String args[]) {
        MyThread thread1 = new MyThread("Thread-1");
        MyThread thread2 = new MyThread("Thread-2");

        thread1.start(); // Start executing run() in a new thread
        thread2.start(); // Start executing run() in a new thread
    }
}
```

Output will show interleaving of "Thread-1" and "Thread-2" messages, demonstrating concurrent execution.

- **Standard Example: Producer-Consumer Problem:**
  - **Concept:** A classic concurrency problem. It involves two types of processes/threads:
    - **Producer:** Generates data/items and puts them into a shared buffer (or queue).
    - **Consumer:** Takes data/items from the shared buffer and processes them.
  - **Challenge:** Ensuring that the producer doesn't try to add data to a full buffer and the consumer doesn't try to remove data from an empty buffer. Also, ensuring thread-safe access to the shared buffer.
  - **Solution using `wait()` and `notify()` (or `notifyAll()`):**
    - **`wait()`:** A thread calls `wait()` when it needs to pause its execution because a certain condition is not met (e.g., consumer trying to read from empty buffer, producer trying to write to full buffer). It releases the lock on the object and goes into a waiting state.
    - **`notify()`/`notifyAll()`:** A thread calls `notify()` (or `notifyAll()`) when it changes a condition that a waiting thread might be waiting for. It wakes up one (or all) waiting threads, which then re-acquire the lock and check the condition again.
  - **Code Example (Simplified):**

```
import java.util.LinkedList;
import java.util.Queue;

class SharedBuffer {
    private Queue<Integer> buffer = new LinkedList<>();
    private final int CAPACITY = 5;

    // Method for Producer to add item
    public void produce(int item) throws InterruptedException {
        synchronized (this) {
            while (buffer.size() == CAPACITY) {
```

```
                System.out.println("Buffer is full. Producer
waiting...");
                wait(); // Producer waits if buffer is full
            }
            buffer.add(item);
            System.out.println("Produced: " + item + ". Buffer size: "
+ buffer.size());
            notifyAll(); // Notify consumers that buffer has items
        }
    }

    // Method for Consumer to consume item
    public int consume() throws InterruptedException {
        synchronized (this) {
            while (buffer.isEmpty()) {
                System.out.println("Buffer is empty. Consumer
waiting...");
                wait(); // Consumer waits if buffer is empty
            }
            int item = buffer.remove();
            System.out.println("Consumed: " + item + ". Buffer size: "
+ buffer.size());
            notifyAll(); // Notify producers that buffer has space
            return item;
        }
    }
}

class Producer extends Thread {
    private SharedBuffer buffer;
    public Producer(SharedBuffer buffer) { this.buffer = buffer; }
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                buffer.produce(i);
                Thread.sleep(100); // Simulate work
            } catch (InterruptedException e) {
Thread.currentThread().interrupt(); }
        }
    }
}

class Consumer extends Thread {
    private SharedBuffer buffer;
    public Consumer(SharedBuffer buffer) { this.buffer = buffer; }
    public void run() {
        for (int i = 0; i < 10; i++) {
            try {
                buffer.consume();
                Thread.sleep(200); // Simulate work
            } catch (InterruptedException e) {
Thread.currentThread().interrupt(); }
        }
    }
```

```
}

public class ProducerConsumerProblem {
    public static void main(String[] args) {
        SharedBuffer buffer = new SharedBuffer();
        Producer producer = new Producer(buffer);
        Consumer consumer = new Consumer(buffer);

        producer.start();
        consumer.start();
    }
}
```

This example demonstrates how `wait()` and `notifyAll()` are used to coordinate between producer and consumer threads to prevent buffer underflow/overflow and ensure proper synchronization.

4. How to create new threads? Explain "implements Runnable" vs "extends Thread".

In Java, there are two primary ways to create new threads:

1. **By Extending the `java.lang.Thread` Class:**
    - **How to create:**
        - Create a new class that `extends Thread`.
        - Override the `run()` method with the code that the thread should execute.
        - Create an instance of your custom thread class.
        - Call the `start()` method on that instance.
    - **Example:**

```
class MyThreadClass extends Thread {
    public void run() {
        System.out.println("Thread created by extending Thread.");
        // Thread's execution logic here
    }
}
// Usage:
// MyThreadClass t1 = new MyThreadClass();
// t1.start();
```

2. **By Implementing the `java.lang.Runnable` Interface:**
    - **How to create:**
        - Create a new class that `implements Runnable`.
        - Implement the `run()` method with the code that the thread should execute.
        - Create an instance of your `Runnable` implementation.
        - Create a `Thread` object by passing your `Runnable` instance to its constructor.
        - Call the `start()` method on the `Thread` object.
    - **Example:**

```
class MyRunnableClass implements Runnable {
    public void run() {
        System.out.println("Thread created by implementing Runnable.");
        // Thread's execution logic here
    }
}
// Usage:
// MyRunnableClass runnable = new MyRunnableClass();
// Thread t2 = new Thread(runnable);
// t2.start();
```

Difference between "implements Runnable" vs "extends Thread":

Feature	`extends Thread`	`implements Runnable`
Inheritance	The class *is a* `Thread`.	The class defines *what a thread should run*. It *is not* a `Thread`.
Java Inheritance	Java does not support multiple inheritance of classes. If you extend `Thread`, you cannot extend any other class.	You can still extend another class while implementing `Runnable`. This offers more flexibility.
Resource Sharing	Each `Thread` object has its own separate instance, so sharing data among threads can be more complex without explicit mechanisms.	Multiple `Thread` objects can share the *same* `Runnable` instance, making it easier to share data between threads.
Coupling	Tightly couples the task to the thread mechanism.	Decouples the task (the `run()` method) from the thread creation/management mechanism.
API Flexibility	Less flexible if you need to perform other tasks or inherit from another class.	More flexible, as it aligns with the single responsibility principle: your class's primary purpose might not be "being a thread," but "providing a runnable task."
Recommended Approach	Generally **less preferred** for defining task logic.	Generally **more preferred** and recommended for defining tasks that can be executed by a thread.

In summary: `implements Runnable` is generally preferred because it promotes better object-oriented design by separating the task from the thread itself. It allows your class to extend another class, making your design more flexible.

5. What is Executor Framework?

- **Executor Framework (java.util.concurrent package):** The Executor Framework is a set of interfaces and classes in Java's Concurrency API that provides a standardized way to manage and execute asynchronous tasks. It decouples task submission from task execution, abstracting away the complexities of thread creation, lifecycle management, and pooling.

- **Need/Purpose:**
    - **Thread Management:** Manages a pool of worker threads, reducing the overhead of creating and destroying threads for each task.
    - **Task Submission:** Provides a clean interface for submitting tasks (either `Runnable` or `Callable`).
    - **Asynchronous Execution:** Allows tasks to run in the background without blocking the main thread.
    - **Resource Management:** Prevents resource exhaustion by limiting the number of concurrently running threads.
    - **Future Results:** Allows obtaining results from asynchronous tasks using `Future` objects.
    - **Scheduled Execution:** Supports scheduling tasks for future or periodic execution.
- **Key Components:**
    1. `Executor` **Interface:** A simple interface with a single method `void execute(Runnable command)`. It's the base for all executors.
    2. `ExecutorService` **Interface:** An extension of `Executor` that provides more sophisticated features for managing the lifecycle of tasks and the executor itself. It includes methods for submitting `Callable` tasks (which return a result) and for gracefully shutting down the service.
    3. `Executors` **Class:** A utility class that provides factory methods for creating various types of `ExecutorService` instances, such as:
        - `newFixedThreadPool(int nThreads)`: Creates a thread pool with a fixed number of threads.
        - `newCachedThreadPool()`: Creates a thread pool that creates new threads as needed, but reuses existing threads when they are available.
        - `newSingleThreadExecutor()`: Creates an executor that uses a single worker thread.
        - `newScheduledThreadPool(int corePoolSize)`: Creates an executor that can schedule commands to run after a given delay, or to execute periodically.
    4. `Future` **Interface:** Represents the result of an asynchronous computation. It provides methods to check if the computation is complete, wait for its completion, and retrieve the result (`get()`).
    5. `Callable` **Interface:** Similar to `Runnable`, but its `call()` method can return a result and throw a checked exception.
- **Example:**

```
import java.util.concurrent.*;

public class ExecutorFrameworkExample {
    public static void main(String[] args) throws InterruptedException,
ExecutionException {

        // 1. Create an ExecutorService (e.g., a fixed-size thread pool)
        ExecutorService executor = Executors.newFixedThreadPool(2); // Two
worker threads

        // 2. Submit Runnable tasks (don't return a result)
        executor.execute(() -> {
            System.out.println("Runnable Task 1 executed by: " +
Thread.currentThread().getName());
```

```
        });
        executor.execute(() -> {
            System.out.println("Runnable Task 2 executed by: " +
Thread.currentThread().getName());
        });

        // 3. Submit Callable tasks (return a result)
        Future<Integer> future1 = executor.submit(() -> {
            System.out.println("Callable Task 1 executing by: " +
Thread.currentThread().getName());
            TimeUnit.SECONDS.sleep(1); // Simulate work
            return 100;
        });

        Future<String> future2 = executor.submit(new Callable<String>() {
            @Override
            public String call() throws Exception {
                System.out.println("Callable Task 2 executing by: " +
Thread.currentThread().getName());
                return "Hello from Callable!";
            }
        });

        // 4. Retrieve results (blocking call)
        System.out.println("Result of Callable 1: " + future1.get()); //
Blocks until result is ready
        System.out.println("Result of Callable 2: " + future2.get());

        // 5. Shut down the executor (important to release resources)
        executor.shutdown(); // Initiates an orderly shutdown
        // Optional: Wait for all tasks to complete
        if (executor.awaitTermination(5, TimeUnit.SECONDS)) {
            System.out.println("All tasks completed.");
        } else {
            System.out.println("Some tasks did not complete within
timeout.");
        }
    }
}
```

The Executor Framework greatly simplifies concurrent programming in Java by handling the low-level details of thread management.

6. `start()` vs `run()` in threads. What happens if you call `run()` instead of `start()` for an object of class that extends Thread?

- `start()` **Method:**
  - **Purpose:** This method is used to begin the execution of a thread.
  - **Behavior:** When `start()` is called, the Java Virtual Machine (JVM) performs several essential actions:
    1. It creates a **new thread** of execution.

2. It allocates necessary resources for the new thread.
3. It adds the new thread to the thread scheduler's queue, making it eligible to be run by the CPU.
4. Crucially, it then calls the `run()` method *in this newly created thread*.

- **Outcome:** The `start()` method ensures true multithreading, where the `run()` method's code executes concurrently with the calling thread (e.g., the `main` thread).

- **`run()` Method:**
  - **Purpose:** This method contains the actual code that the thread will execute. It defines the "task" that the thread performs.
  - **Behavior:** When `run()` is called directly (like any other regular method call), it simply executes the code within the `run()` method **in the current thread of execution**.
  - **Outcome:** Calling `run()` directly does *not* create a new thread. The code inside `run()` executes sequentially, blocking the calling thread until it completes, just like any other method call. It completely defeats the purpose of multithreading.

- **What happens if you call `run()` instead of `start()` for an object of class that extends `Thread`?** If you have a class that extends `Thread` and you call its `run()` method directly instead of `start()`, the following happens:

1. **No New Thread:** A new thread of execution is **not created**.
2. **Sequential Execution:** The code inside the `run()` method will execute immediately on the **current thread** (the thread from which `run()` was called, typically the `main` thread).
3. **Blocking:** The calling thread will be blocked until the `run()` method completes its execution.
4. **No Multithreading:** The program will behave like a single-threaded application, defeating the entire purpose of using threads.

Example:

```
class MyRunnableTask implements Runnable {
    private String name;
    public MyRunnableTask(String name) { this.name = name; }
    @Override
    public void run() {
        for (int i = 0; i < 3; i++) {
            System.out.println(name + ": " + i + " (Thread: " +
Thread.currentThread().getName() + ")");
            try { Thread.sleep(100); } catch (InterruptedException e) {}
        }
    }
}

public class StartVsRun {
    public static void main(String[] args) {
        MyRunnableTask task1 = new MyRunnableTask("Task-1");
        MyRunnableTask task2 = new MyRunnableTask("Task-2");

        System.out.println("Calling start() for task1...");
        new Thread(task1).start(); // This creates a new thread
```

```
        System.out.println("Calling run() directly for task2...");
        task2.run(); // This executes in the main thread

        System.out.println("Main thread finished its immediate tasks.");
    }
}
```

Expected Output: (Order might vary for `start()` portion, but `run()` will execute entirely before "Main thread finished...")

```
Calling start() for task1...
Calling run() directly for task2...
Task-2: 0 (Thread: main)
Task-2: 1 (Thread: main)
Task-2: 2 (Thread: main)
Main thread finished its immediate tasks.
Task-1: 0 (Thread: Thread-0)
Task-1: 1 (Thread: Thread-0)
Task-1: 2 (Thread: Thread-0)
```

Notice how `Task-2` completes fully within the `main` thread before "Main thread finished..." is printed, whereas `Task-1` runs concurrently with the `main` thread (and `Task-2`'s execution).

7. What is the use of Garbage Collector in Java language?

- **Garbage Collector (GC) in Java:** The Garbage Collector in Java is an automatic memory management system that frees up memory by destroying objects that are no longer referenced by any part of the program. It's a crucial component of the Java Virtual Machine (JVM).
- **Primary Uses/Advantages:**
    1. **Automatic Memory Management:** The most significant use is to automatically manage memory. Developers don't have to explicitly allocate and deallocate memory (like `malloc()` and `free()` in C/C++). This greatly simplifies memory management and reduces programming errors.
    2. **Prevents Memory Leaks:** By reclaiming memory from unreachable objects, the GC helps prevent memory leaks, where memory is allocated but never freed, leading to a slow but continuous depletion of available memory.
    3. **Enhances Program Reliability:** Reduces common programming errors related to memory management, such as:
        - **Dangling Pointers:** Pointers that point to memory locations that have been deallocated.
        - **Double Freeing:** Attempting to deallocate memory that has already been freed.
        - **Memory Corruption:** Errors caused by incorrectly managing memory.
    4. **Improves Performance (indirectly):** While GC itself introduces pauses, by preventing leaks and managing memory efficiently, it ensures that the application has enough available memory for new object creation, which contributes to overall system stability and performance in the long run.

5. **Simplifies Development:** Developers can focus on the application logic rather than low-level memory handling.

- **How it works (High-Level):** The GC primarily works by identifying "reachable" and "unreachable" objects.
    - **Root Objects:** Objects that are always accessible (e.g., objects referenced by active threads, static fields, local variables, JNI references).
    - **Reachable Objects:** Objects that can be reached from a root object by following a chain of references.
    - **Unreachable Objects:** Objects that cannot be reached from any root object. These are considered "garbage" and are eligible for collection.

  The GC then reclaims the memory used by unreachable objects. Java uses various GC algorithms (e.g., Serial, Parallel, CMS, G1, Shenandoah, ZGC) which employ different strategies for marking, sweeping, and compacting memory to optimize performance and minimize pauses.

8. Can you overload the `main()` method?

- **Yes, you can overload the `main()` method in Java.**
- However, only the method with the exact signature `public static void main(String[] args)` is recognized by the Java Virtual Machine (JVM) as the entry point for program execution.
- **Example:**

```
public class MainOverload {

    // The JVM entry point
    public static void main(String[] args) {
        System.out.println("This is the original main() method (JVM entry
point).");
        main(10);             // Calling overloaded main with an int
        main("Hello", "World"); // Calling overloaded main with two Strings
    }

    // Overloaded main method 1 (different parameter type and count)
    public static void main(int num) {
        System.out.println("Overloaded main() with an integer: " + num);
    }

    // Overloaded main method 2 (different parameter count and types)
    public static void main(String s1, String s2) {
        System.out.println("Overloaded main() with two strings: " + s1 + " "
+ s2);
    }

    // Another overloaded main method example
    public void main(double d) { // This is NOT static, so cannot be an
entry point.
        System.out.println("Non-static overloaded main() with a double: " +
d);
```

```
    }
}
```

- **Output when run:**

```
This is the original main() method (JVM entry point).
Overloaded main() with an integer: 10
Overloaded main() with two strings: Hello World
```

As you can see, the JVM always starts execution from `public static void main(String[] args)`. Other overloaded `main` methods can exist, but they must be explicitly called from within the program (e.g., from the actual `main` method or another part of the code). They are treated as regular overloaded methods.

9. How to create thread-safe singleton pattern?

Creating a thread-safe Singleton pattern ensures that only one instance of the class is created even in a multithreaded environment. Here are common approaches in Java:

1. **Synchronized `getInstance()` Method (Eager Initialization - Less Efficient):** The simplest way, but incurs performance overhead because synchronization is applied every time the method is called, even after the instance is created.

```
public class SingletonSynchronizedMethod {
    private static SingletonSynchronizedMethod instance;

    private SingletonSynchronizedMethod() {}

    public static synchronized SingletonSynchronizedMethod getInstance() {
        if (instance == null) {
            instance = new SingletonSynchronizedMethod();
        }
        return instance;
    }
}
```

2. **Double-Checked Locking (DCL) with `volatile` (Recommended for Lazy Initialization):** This approach tries to be lazy and thread-safe without the performance overhead of full method synchronization. `volatile` keyword is crucial to ensure visibility of `instance` variable across threads and prevent reordering of instructions during object creation.

```
public class SingletonDCL {
    private static volatile SingletonDCL instance; // 'volatile' ensures
visibility and prevents reordering

    private SingletonDCL() {}
```

```
    public static SingletonDCL getInstance() {
        if (instance == null) { // First check (no lock)
            synchronized (SingletonDCL.class) { // Acquire lock for instance
creation
                if (instance == null) { // Second check (inside lock)
                    instance = new SingletonDCL();
                }
            }
        }
        return instance;
    }
}
```

3. **Bill Pugh Singleton Implementation (Initialization-on-demand holder idiom):** This is often considered the best approach for lazy initialization in Java as it leverages the JVM's class-loading mechanism to guarantee thread-safety without explicit synchronization.

```
public class SingletonBillPugh {
    private SingletonBillPugh() {}

    // Inner static helper class
    private static class SingletonHelper {
        private static final SingletonBillPugh INSTANCE = new
SingletonBillPugh();
    }

    public static SingletonBillPugh getInstance() {
        return SingletonHelper.INSTANCE; // Instance is created only when
getInstance() is called for the first time
    }
}
```

    - **How it works:** The inner `SingletonHelper` class is not loaded until its `getInstance()` method is called for the first time. Since class loading is inherently thread-safe in Java, the instance creation is guaranteed to be safe and lazy.

4. **Enum Singleton (Most Recommended and Robust):** This is generally considered the best way to implement a Singleton in Java. Enums inherently guarantee that only one instance exists and are thread-safe by default. They also provide serialization out-of-the-box and are immune to reflection attacks.

```
public enum SingletonEnum {
    INSTANCE; // The single instance

    // You can add methods and fields here
    public void doSomething() {
        System.out.println("Singleton operation performed.");
```

```
    }
}
// Usage:
// SingletonEnum.INSTANCE.doSomething();
```

- **Advantages:**
  - **Thread-safe by default:** JVM guarantees that enum constants are initialized safely.
  - **Serialization-safe:** Handles serialization correctly without extra code.
  - **Reflection-safe:** Cannot be instantiated via reflection.
  - **Concise:** Very compact and easy to implement.

10. Can you override the static method?

- **No, you cannot override a static method in Java.**
- **Explanation:**
  - **Static methods belong to the class, not to objects/instances.** They are resolved at compile time based on the reference type (static binding/early binding).
  - **Overriding (runtime polymorphism) depends on dynamic binding**, where the method to be called is determined at runtime based on the actual object type.
  - When you define a static method with the same name and signature in a subclass as in its superclass, it's called **method hiding** or **method shadowing**, not overriding. The subclass's static method hides the superclass's static method.
- **Consequences of Method Hiding:**

```
class Parent {
    public static void staticMethod() {
        System.out.println("Static method from Parent");
    }
    public void instanceMethod() {
        System.out.println("Instance method from Parent");
    }
}

class Child extends Parent {
    // This is method hiding, NOT overriding
    public static void staticMethod() {
        System.out.println("Static method from Child");
    }

    // This is method overriding
    @Override
    public void instanceMethod() {
        System.out.println("Instance method from Child");
    }
}

public class StaticOverrideDemo {
```

```
    public static void main(String[] args) {
        Parent parent = new Parent();
        Parent childAsParent = new Child(); // Upcasting
        Child child = new Child();

        System.out.println("--- Calling staticMethod() ---");
        parent.staticMethod();          // Output: Static method from Parent
        childAsParent.staticMethod();   // Output: Static method from Parent
(resolved at compile-time based on reference type 'Parent')
        child.staticMethod();           // Output: Static method from Child
(resolved at compile-time based on reference type 'Child')
        Parent.staticMethod();          // Output: Static method from Parent
(preferred way to call static methods)
        Child.staticMethod();           // Output: Static method from Child

        System.out.println("\n--- Calling instanceMethod() ---");
        parent.instanceMethod();         // Output: Instance method from
Parent
        childAsParent.instanceMethod();  // Output: Instance method from
Child (runtime polymorphism)
        child.instanceMethod();          // Output: Instance method from
Child
    }
}
```

As seen in the output, `staticMethod()` is called based on the *reference type* (`Parent` or `Child`), not the actual object type, demonstrating that it's compile-time binding (hiding), not runtime binding (overriding). Instance methods, on the other hand, show true polymorphism.

11. Basic OOP principal. Encapsulation, Abstraction explained with good examples.

(This question is a repeat of Q5 under "Object Oriented Concepts". The explanation provided there is comprehensive and can be reused here.)

Encapsulation:

- **Definition:** Bundling of data (attributes) and methods (operations) that operate on the data into a single unit (class). It also involves restricting direct access to some of an object's components, typically by making attributes `private` and providing `public` getter/setter methods.
- **Goal:** Data hiding and controlled access.
- **Good Example (Java): A Bank Account Class**

```
public class BankAccount {
    private String accountNumber; // Data (attributes) are private
    private double balance;       // Data (attributes) are private

    public BankAccount(String accountNumber, double initialBalance) {
        this.accountNumber = accountNumber;
        // Balance is set via a controlled method, not directly
```

```
        if (initialBalance >= 0) {
            this.balance = initialBalance;
        } else {
            this.balance = 0;
            System.out.println("Initial balance cannot be negative. Set to
0.");
        }
    }

    // Getter method for controlled read access
    public String getAccountNumber() {
        return accountNumber;
    }

    // Getter method for controlled read access
    public double getBalance() {
        return balance;
    }

    // Setter-like method for controlled write/modification access
    public void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            System.out.println(amount + " deposited. New balance: " +
balance);
        } else {
            System.out.println("Deposit amount must be positive.");
        }
    }

    // Setter-like method for controlled write/modification access with
logic
    public void withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
            System.out.println(amount + " withdrawn. New balance: " +
balance);
        } else {
            System.out.println("Invalid withdrawal amount or insufficient
funds.");
        }
    }

    // No direct public access to 'balance' or 'accountNumber' fields.
    // All interactions are through the public methods.
}

// Usage:
// BankAccount myAccount = new BankAccount("12345", 1000.0);
// myAccount.deposit(500); // Valid operation
// myAccount.withdraw(200); // Valid operation
// myAccount.balance = -500; // NOT POSSIBLE due to encapsulation (private
field)
// System.out.println(myAccount.getBalance()); // Controlled access
```

This example clearly shows how `accountNumber` and `balance` are encapsulated (private) and can only be interacted with through public methods (`deposit`, `withdraw`, `getAccountNumber`, `getBalance`), ensuring data integrity and preventing invalid states.

Abstraction:

- **Definition:** The process of hiding the complex implementation details and showing only the necessary and essential features of an object or system to the user. It focuses on "what" an object does rather than "how" it does it.
- **Goal:** Simplicity, managing complexity.
- **Good Example (Java): A Remote Control for a TV/Device**

```
// Abstract class representing a generic device that can be controlled
abstract class RemoteControlDevice {
    protected boolean isOn;
    protected int volume;

    public RemoteControlDevice() {
        this.isOn = false;
        this.volume = 0;
    }

    // Abstract methods - concrete devices MUST implement these
    public abstract void turnOn();
    public abstract void turnOff();
    public abstract void changeChannel(int channel);
    public abstract void adjustVolume(int level);

    // Concrete method - common functionality
    public void displayStatus() {
        System.out.println("Device is " + (isOn ? "ON" : "OFF") + ", Volume:
" + volume);
    }
}

// Concrete TV implementation
class Television extends RemoteControlDevice {
    @Override
    public void turnOn() {
        if (!isOn) {
            isOn = true;
            System.out.println("TV is turned ON.");
        }
    }

    @Override
    public void turnOff() {
        if (isOn) {
```

```
            isOn = false;
            System.out.println("TV is turned OFF.");
        }
    }

    @Override
    public void changeChannel(int channel) {
        if (isOn) {
            System.out.println("TV channel changed to: " + channel);
        } else {
            System.out.println("TV is off. Cannot change channel.");
        }
    }

    @Override
    public void adjustVolume(int level) {
        if (isOn) {
            this.volume = Math.max(0, Math.min(100, level)); // Volume
between 0 and 100
            System.out.println("TV volume set to: " + this.volume);
        } else {
            System.out.println("TV is off. Cannot adjust volume.");
        }
    }
}

// Concrete Sound System implementation
class SoundSystem extends RemoteControlDevice {
    @Override
    public void turnOn() { /* Specific sound system logic */
System.out.println("Sound system turned ON."); isOn = true; }
    @Override
    public void turnOff() { /* Specific sound system logic */
System.out.println("Sound system turned OFF."); isOn = false; }
    @Override
    public void changeChannel(int channel) { System.out.println("Sound
system does not have channels."); }
    @Override
    public void adjustVolume(int level) { /* Specific sound system logic */
System.out.println("Sound system volume set to: " + level); this.volume =
level; }
}

public class AbstractionExample {
    public static void main(String[] args) {
        RemoteControlDevice myTv = new Television(); // User interacts with
abstract type
        myTv.turnOn();
        myTv.adjustVolume(50);
        myTv.changeChannel(7);
        myTv.displayStatus();
        myTv.turnOff();

        System.out.println("\n--- Sound System ---");
```

```
        RemoteControlDevice mySoundSystem = new SoundSystem();
        mySoundSystem.turnOn();
        mySoundSystem.adjustVolume(80);
        mySoundSystem.changeChannel(1); // Will show appropriate message for
SoundSystem
        mySoundSystem.displayStatus();
        mySoundSystem.turnOff();
    }
}
```

Here, the `RemoteControlDevice` abstract class defines the common interface (turnOn, turnOff, changeChannel, adjustVolume) for *any* remote-controlled device, hiding the underlying complexities of how a TV or a Sound System actually implements these actions. The user (client code) interacts with the abstract `RemoteControlDevice` type, not knowing or caring about the specific internal mechanics of each concrete device.

12. If multiple interfaces have the same method declaration, does it cause name collision or any other problem when they are implemented in a class?

- **No, it does not cause a name collision or any other problem** when a class implements multiple interfaces that declare the same method (with the exact same signature: name, parameters, and return type).
- **Explanation:**
    - Interfaces in Java primarily define **contracts** or **type definitions**. They declare *what* a class should do, but not *how* (before Java 8's default methods).
    - When a class implements multiple interfaces with identical method signatures, it means the class is simply committing to fulfill the same contract requirements from different sources.
    - The implementing class provides a **single concrete implementation** for that method, and this single implementation satisfies the contract of all interfaces that declare it. The compiler (and JVM) has no ambiguity because there's only one actual method body to execute.
- **Example:**

```
interface InterfaceA {
    void commonMethod();
    void methodA();
}

interface InterfaceB {
    void commonMethod(); // Same method signature as in InterfaceA
    void methodB();
}

class MyClass implements InterfaceA, InterfaceB {
    @Override
    public void commonMethod() {
        System.out.println("Implementation of commonMethod from both
InterfaceA and InterfaceB.");
```

```
    }

    @Override
    public void methodA() {
        System.out.println("Implementation of methodA from InterfaceA.");
    }

    @Override
    public void methodB() {
        System.out.println("Implementation of methodB from InterfaceB.");
    }

    public static void main(String[] args) {
        MyClass obj = new MyClass();
        obj.commonMethod();
        obj.methodA();
        obj.methodB();

        // You can also reference it by interface types
        InterfaceA objA = obj;
        objA.commonMethod();
        objA.methodA();

        InterfaceB objB = obj;
        objB.commonMethod();
        objB.methodB();
    }
}
```

What if interfaces have `default` methods with the same signature (Java 8+)? This *does* cause a compile-time error due to ambiguity. In such a scenario, the implementing class *must* provide its own explicit implementation of the method to resolve the conflict.

```
interface InterfaceX {
    default void doSomething() {
        System.out.println("Default from InterfaceX");
    }
}

interface InterfaceY {
    default void doSomething() {
        System.out.println("Default from InterfaceY");
    }
}

class MyConflictingClass implements InterfaceX, InterfaceY {
    // Compile-time error if 'doSomething' is not overridden:
    // "Error: class MyConflictingClass inherits unrelated defaults for
doSomething()
    // from types InterfaceX and InterfaceY"
```

```
    @Override
    public void doSomething() { // Must provide explicit implementation to
resolve conflict
        System.out.println("MyConflictingClass's own implementation of
doSomething.");
        InterfaceX.super.doSomething(); // Optionally call a specific
default
    }
}
```

So, for purely abstract method declarations, no issue. For `default` methods with the same signature, explicit overriding is required.

13. What is Upcasting and Downcasting in Java? Where is it used in Java? Explain with examples.

(This question is a repeat of Q11 under "Object Oriented Concepts". The explanation provided there is comprehensive and can be reused here.)

Upcasting:

- **Definition:** Assigning a reference of a subclass type to a variable of its superclass type. It's implicitly done by the compiler and is always safe.
- **Purpose:** To achieve polymorphism. It allows you to treat objects of different subclasses uniformly as objects of their common superclass, enabling generic code that works with a hierarchy of classes.
- **Example:**

```
class Vehicle {
    void startEngine() { System.out.println("Vehicle engine started."); }
}
class Car extends Vehicle {
    @Override
    void startEngine() { System.out.println("Car engine started with key.");
}
    void honk() { System.out.println("Honk honk!"); }
}

public class CastingExample {
    public static void main(String[] args) {
        Vehicle myCar = new Car(); // Upcasting: A Car object is referenced
by a Vehicle type variable
        myCar.startEngine(); // Calls Car's startEngine() due to
polymorphism

        // myCar.honk(); // Compile-time error: 'honk()' is not defined in
Vehicle
                         // Even though 'myCar' points to a Car object, the
compiler only sees the Vehicle methods.
    }
}
```

Downcasting:

- **Definition:** Converting a reference of a superclass type to a reference of its subclass type. It must be explicitly cast by the programmer and can lead to `ClassCastException` at runtime if the object being cast is not actually an instance of the target subclass.
- **Purpose:** To access methods or attributes that are specific to the subclass, after an object has been upcasted.
- **Where it's used:**
  - **Accessing specialized behavior:** When you know a general type reference actually points to a specific subclass object, and you need to call a method unique to that subclass.
  - **Working with collections of mixed types:** When you have a collection (e.g., `List<Animal>`) that contains objects of different actual types (`Dog`, `Cat`), and you need to perform type-specific operations.
- **Example:**

```
// (Continuing from previous example)
// Vehicle myVehicle = new Car(); // myVehicle is a Vehicle reference to a
Car object

// Downcasting to access Car-specific method
if (myVehicle instanceof Car) { // Safe check before downcasting
    Car carObject = (Car) myVehicle; // Explicit downcast
    carObject.honk(); // Now we can call 'honk()'
    carObject.startEngine(); // Still calls Car's startEngine()
} else {
    System.out.println("myVehicle is not a Car.");
}

Vehicle anotherVehicle = new Vehicle(); // A plain Vehicle object
// Try to downcast a Vehicle object to Car (will fail at runtime)
try {
    Car notACar = (Car) anotherVehicle; // ClassCastException will occur
here
    notACar.honk();
} catch (ClassCastException e) {
    System.out.println("Error: Cannot cast Vehicle to Car. " +
e.getMessage());
}
```

In summary:

- **Upcasting:** Safe, implicit, narrows access (to superclass methods). Used for polymorphism.
- **Downcasting:** Potentially unsafe, explicit, broadens access (to subclass methods). Used to regain specialized behavior after upcasting, always requires `instanceof` check for safety.

14. Explain default methods in the interfaces.

- **Default Methods (Defender Methods/Virtual Extension Methods):**
  - **Introduction:** Introduced in Java 8.
  - **Purpose:** To allow adding new methods to interfaces *without breaking existing implementations* of those interfaces. Before Java 8, adding a new method to an interface would require all implementing classes to provide an implementation, breaking backward compatibility.
  - **Mechanism:** Default methods have a method body (an implementation) directly within the interface itself. They are declared with the `default` keyword.
  - **Behavior:**
    - Implementing classes automatically inherit the default implementation.
    - Implementing classes can **override** the default implementation if they need specific behavior.
    - They are `public` implicitly.
    - They cannot access instance fields (non-static fields) of the implementing class, but they can access static methods and fields of the interface.
- **Why they were introduced (Motivation):**
  - **Backward Compatibility:** Primarily to allow evolution of interfaces, especially for core Java APIs (like adding `forEach` to `Iterable` without breaking all existing `Iterable` implementations).
  - **Code Reusability:** Provide common utility methods that can be shared among implementing classes, reducing code duplication.
  - **Multiple Inheritance of Behavior:** While Java doesn't support multiple inheritance of classes, default methods provide a limited form of multiple inheritance of *behavior*.
- **Example:**

```
interface Drawable {
    void draw(); // Abstract method

    // Default method: provides a default implementation
    default void resize() {
        System.out.println("Resizing shape by default implementation.");
    }

    // Static method (also introduced in Java 8 with default methods
concept)
    static void printInfo() {
        System.out.println("This is a Drawable interface.");
    }
}

class Circle implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a Circle.");
    }
    // Circle implicitly inherits the default resize() method
}
```

```
class Square implements Drawable {
    @Override
    public void draw() {
        System.out.println("Drawing a Square.");
    }

    // Square overrides the default resize() method
    @Override
    public void resize() {
        System.out.println("Resizing Square specifically: maintaining aspect
ratio.");
    }
}

public class DefaultMethodExample {
    public static void main(String[] args) {
        Circle circle = new Circle();
        circle.draw();
        circle.resize(); // Uses default implementation from interface

        System.out.println("---");

        Square square = new Square();
        square.draw();
        square.resize(); // Uses overridden implementation in Square class

        // Static methods in interfaces are called directly on the interface
        Drawable.printInfo();
    }
}
```

Output:

```
Drawing a Circle.
Resizing shape by default implementation.
---
Drawing a Square.
Resizing Square specifically: maintaining aspect ratio.
This is a Drawable interface.
```

This demonstrates how `Circle` gets the default `resize()` behavior automatically, while `Square` provides its own specific `resize()` implementation, showing the flexibility default methods provide.

15. How do HashSet and HashMap work internally in Java? What is the role of `equals()` and `hashCode()` methods in this context?

- **Internal Working of `HashMap`:** `HashMap` stores data in key-value pairs. Internally, it is implemented as an **array of `Node` objects** (or `Entry` objects in older Java versions). Each `Node` represents a key-value pair and often points to the next `Node` in a linked list structure to handle **collisions**. From Java 8, if a

bucket contains too many elements (more than `TREEIFY_THRESHOLD`, typically 8), the linked list can be converted into a **balanced tree** (Red-Black Tree) for better performance in worst-case scenarios.

1. **`put(K key, V value)`:**
    - When you call `put(key, value)`, `HashMap` first calculates the `hashCode()` of the `key`.
    - This hash code is then used to determine the **bucket (array index)** where the key-value pair will be stored. The index is typically calculated as `index = key.hashCode() & (capacity - 1)`.
    - If the bucket is empty, the new `Node` is placed there.
    - If the bucket is already occupied (a **collision** occurs), `HashMap` traverses the linked list (or tree) at that bucket.
    - For each `Node` in the list/tree, it compares the new `key` with existing keys using the `equals()` method.
    - If `equals()` returns `true` for an existing key, the old value associated with that key is replaced with the new value.
    - If `equals()` returns `false` for all existing keys in the bucket, the new `Node` is added to the end of the linked list (or as a new node in the tree).
    - If the number of elements in a bucket exceeds a certain threshold (TREEIFY_THRESHOLD), the linked list is converted to a tree for better performance.
    - If the `HashMap`'s size exceeds its `load factor * capacity`, it resizes (rehashes) the internal array to a larger size.
2. **`get(Object key)`:**
    - When you call `get(key)`, `HashMap` again calculates the `hashCode()` of the provided `key` to find the correct bucket.
    - It then traverses the linked list (or tree) at that bucket.
    - For each `Node` in the list/tree, it compares the provided `key` with the `key` stored in the `Node` using the `equals()` method.
    - If `equals()` returns `true`, the corresponding `value` is returned.
    - If the key is not found after traversing the entire bucket, `null` is returned.

- **Internal Working of `HashSet`:** `HashSet` stores unique elements. It is internally implemented using a `HashMap`. When you create a `HashSet`, it internally creates a `HashMap` where:
    - The elements you add to the `HashSet` are stored as **keys** in the internal `HashMap`.
    - A dummy `Object` (a static final `PRESENT` object) is stored as the **value** for every key in the `HashMap`.

    1. **`add(E e)`:**
        - When you call `add(element)` on `HashSet`, it internally calls `map.put(element, PRESENT)`.
        - The `HashMap`'s `put()` method handles the hashing and equality checks. If the element (key) already exists, `HashMap` just returns the old dummy value, effectively preventing the duplicate element from being added.
        - It returns `true` if the element was added successfully (i.e., it was not a duplicate), `false` otherwise.

2. **`contains(Object o)`:**
    - When you call `contains(element)` on `HashSet`, it internally calls `map.containsKey(element)`.
    - The `HashMap`'s `containsKey()` method performs the hash and equality checks to determine if the element (key) exists.

- **Role of `equals()` and `hashCode()` Methods:**

  The correct implementation of `equals()` and `hashCode()` methods is absolutely critical for the proper functioning and performance of `HashMap`, `HashSet`, and other hash-based collections (`Hashtable`, `LinkedHashMap`, `LinkedHashSet`).

  - **`hashCode()`:**
    - **Purpose:** To generate an integer hash value for an object. This hash value is used by hash-based collections to quickly determine which bucket an object belongs to.
    - **Contract:**
      - If two objects are `equals()` according to the `equals()` method, then their `hashCode()` methods *must* produce the same integer result.
      - If two objects are *not* `equals()`, their `hashCode()` methods are *not required* to produce different integer results (collisions are allowed). However, a good `hashCode()` implementation minimizes collisions to improve performance.
    - **Impact of Poor `hashCode()`:** If `hashCode()` is not overridden (or poorly implemented) for custom objects used as keys/elements, all objects might hash to the same bucket, degrading performance to O(n) (like a linked list) for `put` and `get` operations.

  - **`equals(Object obj)`:**
    - **Purpose:** To determine if two objects are logically equal (represent the same data or entity).
    - **Contract:**
      - **Reflexive:** `x.equals(x)` must be `true`.
      - **Symmetric:** If `x.equals(y)` is `true`, then `y.equals(x)` must be `true`.
      - **Transitive:** If `x.equals(y)` is `true` and `y.equals(z)` is `true`, then `x.equals(z)` must be `true`.
      - **Consistent:** Multiple invocations of `x.equals(y)` must consistently return the same result, provided no information used in `equals` comparisons on the objects is modified.
      - For any non-null reference value `x`, `x.equals(null)` must return `false`.
    - **Impact of Poor `equals()`:** If `equals()` is not overridden (or poorly implemented), `HashMap` might store duplicate keys or fail to retrieve existing values, as it relies on `equals()` to confirm object identity within a bucket after hashing.

  Together: `hashCode()` quickly identifies the probable bucket, and `equals()` then precisely identifies the correct element within that bucket. If these methods are not implemented consistently with their contracts, hash-based collections will behave incorrectly (e.g., duplicates in `HashSet`, inability to retrieve elements from `HashMap`).

16. What is their initial capacity of a collection and what happens when it is exhausted? Explain with examples.

- **Initial Capacity:** The initial capacity of a collection refers to the number of elements it can hold without resizing its internal data structure. It's the default size allocated when the collection is first created without specifying a capacity.
- **What happens when it is exhausted (Resizing):** When a dynamic collection (like `ArrayList` or `HashMap`) reaches its capacity and you try to add more elements, it needs to increase its internal storage. This process involves:
    1. **Creating a new, larger internal array/table.**
    2. **Copying all existing elements** from the old structure to the new, larger structure.
    3. For `HashMap` and `HashSet`, this copying process is called **rehashing**, as elements are not just copied but re-evaluated for their new bucket index based on the new capacity.
    4. **Discarding the old, smaller structure.**

    This resizing operation is **costly** (O(n) for `ArrayList`, O(n) for `HashMap` where n is current size) because it involves memory allocation and copying. Therefore, it's often recommended to initialize collections with a reasonable initial capacity if the expected size is known, to minimize resizings.
- **Examples:**
    1. **`ArrayList`:**
        - **Default Initial Capacity:** `10`
        - **Resizing Behavior:** When an `ArrayList`'s capacity is exhausted, its new capacity is usually `current_capacity + (current_capacity / 2)` (i.e., 1.5 times the current capacity).
        - **Example:**

```
import java.util.ArrayList;

public class ArrayListCapacityExample {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>(); // Default
capacity is 10
        System.out.println("Initial capacity (not directly
accessible, but effectively 0 until first add, then 10):");
        // Can't directly get capacity, but internal array size is
10 after first add

        for (int i = 0; i < 10; i++) {
            list.add("Element " + i);
        }
        System.out.println("Size after 10 elements: " +
list.size()); // Size is 10

        list.add("Element 10"); // This will trigger resizing
        System.out.println("Size after 11 elements: " +
list.size()); // Size is 11
```

```
        // Internal capacity would have resized from 10 to 10 +
10/2 = 15.
        // All 10 existing elements are copied to the new array.

        list.add("Element 11");
        list.add("Element 12");
        list.add("Element 13");
        list.add("Element 14"); // Now at capacity 15
(hypothetically)
        System.out.println("Size after 15 elements: " +
list.size());

        list.add("Element 15"); // This will trigger another
resizing
        System.out.println("Size after 16 elements: " +
list.size());
        // Internal capacity would now be 15 + 15/2 = 22 (integer
division).
    }
}
```

2. `HashMap`:

- **Default Initial Capacity:** `16` (must be a power of 2 for efficient hashing).
- **Default Load Factor:** `0.75f`. This means `HashMap` will resize when its size reaches `capacity * load_factor`. For a default capacity of 16, it will resize when it has `16 * 0.75 = 12` entries.
- **Resizing Behavior:** When `HashMap`'s size exceeds `capacity * load_factor`, it doubles its capacity and **rehashes** all existing entries into the new, larger table.
- **Example:**

```
import java.util.HashMap;

public class HashMapCapacityExample {
    public static void main(String[] args) {
        HashMap<Integer, String> map = new HashMap<>(); // Default
capacity 16, load factor 0.75
        System.out.println("Map created. Initial threshold for
resize (16 * 0.75) = 12");

        for (int i = 0; i < 12; i++) {
            map.put(i, "Value " + i);
        }
        System.out.println("Size after 12 puts: " + map.size());
// Size is 12

        map.put(12, "Value 12"); // This will trigger resizing
(threshold was 12)
```

```
        System.out.println("Size after 13 puts: " + map.size());
// Size is 13
        // Internal capacity would have resized from 16 to 32.
        // All 12 existing entries are rehashed into the new 32-
sized array.
        // New threshold is 32 * 0.75 = 24.
    }
}
```

`HashSet` behaves similarly as it uses `HashMap` internally.

17. Is HashMap thread safe? How to make it thread safe?

- **Is `HashMap` thread-safe?** No, `HashMap` is **not thread-safe**. If multiple threads access a `HashMap` concurrently and at least one of them modifies the map structure (e.g., `put`, `remove`), it can lead to inconsistent state, corrupted data, or even infinite loops during operations. This is because `HashMap`'s internal operations (like resizing, linking nodes) are not synchronized.
- **How to make it thread-safe?**

  There are several ways to achieve thread-safety for a map in Java:

  1. **Using `Collections.synchronizedMap()`:**
     - **Approach:** This is a wrapper method provided by the `Collections` utility class. It returns a synchronized (thread-safe) `Map` backed by the specified (non-thread-safe) `Map`.
     - **Mechanism:** All methods of the returned map are synchronized (locked on the returned map object).
     - **Drawback:** It provides synchronization at the object level (a single lock for the entire map), which can lead to contention and poor performance in highly concurrent scenarios (only one thread can access the map at a time for any operation).
     - **Example:**

       ```
       import java.util.Collections;
       import java.util.HashMap;
       import java.util.Map;

       Map<String, String> synchronizedHashMap =
       Collections.synchronizedMap(new HashMap<>());
       // All operations on synchronizedHashMap are now thread-safe.
       synchronizedHashMap.put("key", "value");
       String value = synchronizedHashMap.get("key");
       ```

       Note: Iteration over a synchronized map must be manually synchronized by the client code to avoid `ConcurrentModificationException`.

  2. **Using `ConcurrentHashMap` (Preferred for high concurrency):**

- **Approach:** `ConcurrentHashMap` is a highly optimized, thread-safe implementation of the `Map` interface (part of Java's `java.util.concurrent` package).
- **Mechanism:** It uses a technique called **segment-locking** (or fine-grained locking, where different parts of the map are locked independently) or **lock-free algorithms** (optimistic locking using CAS operations in Java 8+) to allow concurrent reads and writes. This drastically reduces contention compared to `synchronizedMap()`.
- **Advantages:**
  - Much better performance under high concurrency compared to `Collections.synchronizedMap()`.
  - Does not throw `ConcurrentModificationException` during iteration if the map is modified concurrently (iterators are "fail-safe").
- **Example:**

```
import java.util.concurrent.ConcurrentHashMap;
import java.util.Map;

Map<String, String> concurrentHashMap = new ConcurrentHashMap<>();
// All operations are thread-safe and highly concurrent.
concurrentHashMap.put("key", "value");
String value = concurrentHashMap.get("key");
```

3. **Using `Hashtable` (Legacy, generally not recommended):**

- **Approach:** `Hashtable` is an older, legacy class (from Java 1.0) that is thread-safe.
- **Mechanism:** Like `Collections.synchronizedMap()`, it achieves thread-safety by synchronizing all its methods using a single intrinsic lock.
- **Disadvantages:**
  - Poor performance under high concurrency due to coarse-grained locking.
  - `HashMap` generally offers better performance in single-threaded environments.
  - `Hashtable` does not allow `null` keys or `null` values, whereas `HashMap` and `ConcurrentHashMap` (since Java 7) do.
- **Conclusion:** Almost always prefer `ConcurrentHashMap` over `Hashtable` for new code requiring thread-safety.

Summary: For thread-safe map operations, `ConcurrentHashMap` is the modern and generally most performant choice for high-concurrency scenarios. `Collections.synchronizedMap()` can be used for simpler cases or when you need to wrap an existing `HashMap`, but with a potential performance bottleneck. Avoid `Hashtable` in new development.

18. Tell me the Difference between StringBuilder and StringBuffer. Why is String immutable? What is a String pool?

- **Difference between `StringBuilder` and `StringBuffer`:**

Feature	`StringBuffer`	`StringBuilder`

Feature	`StringBuffer`	`StringBuilder`
Mutability	Mutable (can change its sequence of characters).	Mutable (can change its sequence of characters).
Thread-Safe	**Yes**, thread-safe. All its public methods are `synchronized`.	**No**, not thread-safe. Methods are not synchronized.
Performance	**Slower** than `StringBuilder` due to synchronization overhead.	**Faster** than `StringBuffer` as it doesn't incur synchronization overhead.
Usage	Preferred in **multithreaded environments** where string manipulation needs to be synchronized.	Preferred in **single-threaded environments** or where external synchronization is handled.
Introduced	Java 1.0	Java 5

In summary: If you need thread-safety, use `StringBuffer`. If performance is critical and you are in a single-threaded environment (or handling synchronization externally), use `StringBuilder`. For most common string concatenations within a single method, `StringBuilder` is the default choice.

- **Why is `String` immutable?** In Java, `String` objects are **immutable**, meaning once a `String` object is created, its content (the sequence of characters) cannot be changed. Any operation that appears to modify a `String` (like concatenation or `substring()`) actually creates a *new* `String` object.

  Reasons for `String` immutability:

  1. **Security:**
     - **Method Parameters:** Strings are frequently used as method parameters (e.g., database connection URLs, file paths, network addresses). If they were mutable, a malicious method could alter the string after it's passed, potentially leading to security vulnerabilities.
     - **Hashing:** Strings are used in data structures like `HashMap` and `HashSet`. If a string's content (and thus its hash code) could change after being put into a map, the map might become corrupted and unable to retrieve the object.
  2. **Thread Safety:**
     - Immutable objects are inherently thread-safe because their state cannot be modified after creation. Multiple threads can share and access a `String` object without any synchronization issues, leading to cleaner and safer concurrent code.
  3. **Caching and Performance (String Pool):**
     - Because strings are immutable, the JVM can implement the "String Pool" (or String Interning) effectively. This allows multiple `String` variables to reference the same `String` object in memory if they have the same literal value, saving memory and improving performance for string comparisons.
  4. **Class Loading:**

    - String is used for class names when Java loads classes. If strings were mutable, dynamically loading a class could lead to security risks if the class name string could be altered by a rogue process.

5. **Hash Code Caching:**

    - An `String` object's `hashCode()` is computed once and cached. Since the string's content never changes, its hash code also remains constant, making it very efficient for use in hash-based collections.

- **What is a String pool?** The String Pool (also known as String Intern Pool or String Constant Pool) is a special area in the Java Heap memory (in PermGen space in older Java versions, now in metaspace/heap itself) where `String` literals are stored.
    - **Purpose:** To save memory and improve performance by storing only one copy of each unique `String` literal.
    - **How it works:**
        - When you create a `String` using a **literal** (e.g., `String s = "hello";`), the JVM first checks if a `String` object with the value "hello" already exists in the String pool.
        - If it exists, the existing object's reference is returned.
        - If it doesn't exist, a new `String` object is created in the pool, and its reference is returned.
        - When you create a `String` using the `new` keyword (e.g., `String s = new String("hello");`), a *new* object is always created in the heap, regardless of whether "hello" exists in the pool or not. You can explicitly "intern" this new string into the pool later using the `intern()` method.
    - **Example:**

```
String s1 = "Java"; // "Java" goes into the String pool if not already
there. s1 refers to it.
String s2 = "Java"; // s2 also refers to the _same_ "Java" object in
the pool.
String s3 = new String("Java"); // A _new_ "Java" object is created on
the heap (outside the pool).
String s4 = s3.intern(); // s4 refers to the "Java" object _from the
pool_.

    System.out.println(s1 == s2);       // true (same object in pool)
    System.out.println(s1 == s3);       // false (s1 from pool, s3 new
on heap)
    System.out.println(s1 == s4);       // true (s1 from pool, s4
forced from pool)
    System.out.println(s3 == s4);       // false (s3 new on heap, s4
from pool)
    ```

The String pool helps optimize memory usage and speeds up string
comparisons, especially when dealing with many identical string
literals.
```
```

19. What is the difference between `==` and `equals()` in Java? Explain in context of primitive values and objects with examples.

The `==` operator and the `equals()` method are both used for comparison in Java, but they serve different purposes and operate differently based on whether they are comparing primitive values or objects.

- **`==` Operator:**
  - **Purpose:**
    - **For Primitive Values:** Compares the actual *values* of the primitives.
    - **For Objects (Non-primitives/References):** Compares the *memory addresses* (references) of the objects. It checks if two object references point to the exact same object in memory.
  - **When to Use:**
    - To compare primitive data types (e.g., `int`, `char`, `boolean`, `double`).
    - To check if two object references point to the *exact same object instance*.
- **`equals()` Method:**
  - **Purpose:**
    - **For Objects:** Compares the *content* or *state* of two objects to determine if they are logically equivalent.
    - **Default Behavior (from `Object` class):** The default implementation of `equals()` in the `Object` class (the root of all Java classes) is identical to the `==` operator for objects; it compares memory addresses.
    - **Overriding:** For most custom classes (and many built-in classes like `String`, `Integer`, `Date`), the `equals()` method is **overridden** to provide a meaningful content-based comparison.
  - **When to Use:**
    - To check if two objects are *logically equal* based on their content, not necessarily if they are the exact same instance.
- **Examples:**

  1. Primitive Values:

```
int num1 = 10;
int num2 = 10;
int num3 = 20;

System.out.println("--- Primitives ---");
System.out.println("num1 == num2: " + (num1 == num2)); // true (values are
equal)
System.out.println("num1 == num3: " + (num1 == num3)); // false (values are
different)
// You cannot call .equals() on primitive types.
```

  2. Objects (Without `equals()` Overriding - Custom Class):

```
class MyClass {
    int value;
    MyClass(int v) { this.value = v; }
}

MyClass obj1 = new MyClass(10);
MyClass obj2 = new MyClass(10); // Different object instance, same value
MyClass obj3 = obj1;            // obj3 refers to the same object as obj1

System.out.println("\n--- Custom Objects (No equals() override) ---");
System.out.println("obj1 == obj2: " + (obj1 == obj2)); // false (different
memory addresses)
System.out.println("obj1.equals(obj2): " + (obj1.equals(obj2))); // false
(default equals() is like ==)
System.out.println("obj1 == obj3: " + (obj1 == obj3)); // true (same memory
address)
System.out.println("obj1.equals(obj3): " + (obj1.equals(obj3))); // true
(default equals() is like ==)
```

3. Objects (With `equals()` Overriding - `String` class):

```
String s1 = "hello";
String s2 = "hello"; // Due to String pool, this points to the same literal
"hello"
String s3 = new String("hello"); // Creates a new object on the heap
String s4 = "world";

System.out.println("\n--- String Objects (equals() overridden) ---");
System.out.println("s1 == s2: " + (s1 == s2));       // true (both refer to
the same object in String pool)
System.out.println("s1.equals(s2): " + (s1.equals(s2))); // true (content is
same)

System.out.println("s1 == s3: " + (s1 == s3));       // false (s1 from pool,
s3 new on heap - different addresses)
System.out.println("s1.equals(s3): " + (s1.equals(s3))); // true (content is
same)

System.out.println("s1 == s4: " + (s1 == s4));       // false (different
content, different objects)
System.out.println("s1.equals(s4): " + (s1.equals(s4))); // false (different
content)
```

This example clearly illustrates why `equals()` is crucial for content comparison for objects, especially when they might have the same value but are distinct instances.

20. What is a fail-fast and fail-safe iterator? Explain with code.

These terms describe the behavior of iterators when the underlying collection is modified during iteration.

- **Fail-Fast Iterators:**
  - **Definition:** Iterators that immediately throw a `ConcurrentModificationException` if the underlying collection is structurally modified (i.e., elements are added, removed, or the order is changed) at any time after the iterator is created, *except* through the iterator's own `remove()` or `add()` (if supported) methods.
  - **Purpose:** To quickly detect and report concurrent modifications, preventing indeterminate behavior or silent data corruption. They are designed for single-threaded use or when external synchronization is explicitly handled.
  - **Mechanism:** Internally, they often use a `modCount` variable in the collection. The iterator stores an expected `modCount`. If `modCount` changes during iteration, it means the collection was modified externally, and the exception is thrown.
  - **Examples:** Iterators returned by most `java.util` collection classes like `ArrayList`, `HashMap`, `HashSet`, `LinkedList`, `Vector`.
- **Fail-Safe Iterators:**
  - **Definition:** Iterators that do *not* throw a `ConcurrentModificationException` if the underlying collection is structurally modified during iteration. They operate on a separate copy or a snapshot of the collection at the time the iterator was created.
  - **Purpose:** To provide a consistent view of the collection even if concurrent modifications occur. They are suitable for scenarios where concurrent modification is expected and needs to be tolerated.
  - **Mechanism:** They typically work on a copy of the collection (e.g., `CopyOnWriteArrayList`) or by using sophisticated internal concurrency control mechanisms (e.g., `ConcurrentHashMap`).
  - **Examples:** Iterators returned by classes in `java.util.concurrent` package like `CopyOnWriteArrayList`, `ConcurrentHashMap`.
- **Explanation with Code:**

  Fail-Fast Iterator Example:

```
import java.util.ArrayList;
import java.util.Iterator;
import java.util.ConcurrentModificationException;

public class FailFastExample {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        Iterator<String> iterator = names.iterator();

        System.out.println("Starting fail-fast iteration...");
        try {
            while (iterator.hasNext()) {
                String name = iterator.next();
                System.out.println("Processing: " + name);
```

```
                // Simulate an external modification to the collection
                if (name.equals("Bob")) {
                    System.out.println("Attempting to modify collection
externally...");
                    names.add("David"); // This structural modification will
cause ConcurrentModificationException
                                        // if iterator.next() is called again.
                }
            }
        } catch (ConcurrentModificationException e) {
            System.out.println("Caught ConcurrentModificationException: " +
e.getMessage());
            System.out.println("Iterator detected external modification and
failed fast.");
        }
        System.out.println("List after attempt: " + names);
    }
}
```

Output:

```
Starting fail-fast iteration...
Processing: Alice
Processing: Bob
Attempting to modify collection externally...
Caught ConcurrentModificationException: null
Iterator detected external modification and failed fast.
List after attempt: [Alice, Bob, Charlie, David]
```

Fail-Safe Iterator Example:

```
import java.util.Iterator;
import java.util.concurrent.CopyOnWriteArrayList; // A fail-safe collection

public class FailSafeExample {
    public static void main(String[] args) {
        CopyOnWriteArrayList<String> names = new CopyOnWriteArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        Iterator<String> iterator = names.iterator();

        System.out.println("Starting fail-safe iteration...");
        while (iterator.hasNext()) {
            String name = iterator.next();
            System.out.println("Processing: " + name);
```

```
            // Simulate a concurrent modification to the collection
            if (name.equals("Bob")) {
                System.out.println("Modifying collection concurrently...");
                names.add("David"); // This modification will NOT cause an
exception
                                    // because the iterator works on a
snapshot.
            }
        }
        System.out.println("Iteration complete.");
        System.out.println("List after modification: " + names);
    }
}
```

Output:

```
Starting fail-safe iteration...
Processing: Alice
Processing: Bob
Modifying collection concurrently...
Processing: Charlie
Iteration complete.
List after modification: [Alice, Bob, Charlie, David]
```

Notice that "David" was added to the list, but it was not seen by the iterator because the iterator was working on a snapshot of the list taken at the time it was created. This is the characteristic behavior of a fail-safe iterator.

21. What is the lambda expression in Java? How data types of lambda arguments and return types are determined?

- **Lambda Expression in Java:**
  - **Introduction:** Introduced in Java 8.
  - **Definition:** A concise way to represent an anonymous function (a function without a name). It allows you to write functional interfaces more compactly and pass behavior as an argument.
  - **Syntax:** `(parameters) -> expression` or `(parameters) -> { statements; }`
  - **Purpose:** Enables functional programming in Java, making code more readable, flexible, and suitable for parallel processing with the Stream API. It's often used for implementing single-method interfaces (functional interfaces).
- **Example:**

```
// Traditional anonymous inner class for Runnable
Runnable oldWay = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running with anonymous inner class.");
```

```
    }
};

// Lambda expression for Runnable
Runnable newWay = () -> System.out.println("Running with lambda
expression.");

oldWay.run();
newWay.run();

// Example with parameters and return value (using a functional interface)
interface Calculator {
    int operate(int a, int b);
}

Calculator add = (x, y) -> x + y;
Calculator subtract = (a, b) -> { return a - b; }; // Block body with return
statement

System.out.println("5 + 3 = " + add.operate(5, 3));
System.out.println("10 - 4 = " + subtract.operate(10, 4));
```

- **How data types of lambda arguments and return types are determined:** The data types of lambda arguments and return types are determined through **Type Inference** by the Java compiler. The compiler uses the **context** in which the lambda expression is used to infer these types. This context is primarily the **functional interface** that the lambda expression is implementing.

    1. **Functional Interface:** A lambda expression can only be assigned to a **functional interface**. A functional interface is an interface that has exactly one abstract method (also known as a Single Abstract Method - SAM). Annotating it with `@FunctionalInterface` is good practice, but not mandatory.
    2. **Inference Process:**
        - The compiler examines the signature of the single abstract method in the target functional interface.
        - It then matches the lambda expression's parameters and return type to this method signature.

    Example (Revisiting `Calculator`):

```
@FunctionalInterface
interface Calculator {
    int operate(int a, int b); // This is the Single Abstract Method (SAM)
}

// Lambda expression: (x, y) -> x + y
Calculator add = (x, y) -> x + y;
// How types are inferred:
// 1. Target type is `Calculator`.
// 2. `Calculator`'s SAM is `int operate(int a, int b)`.
```

```
// 3. The compiler infers that `x` must be `int`, `y` must be `int`.
// 4. The expression `x + y` returns an `int`, which matches the `int`
return type of `operate`.

// Another example with different parameter names and implicit type
inference
List<String> names = Arrays.asList("apple", "banana", "cherry");
names.forEach(s -> System.out.println(s.toUpperCase()));
// How types are inferred:
// 1. `forEach` method expects a `Consumer<String>` functional interface.
// 2. `Consumer<T>` has a SAM: `void accept(T t)`.
// 3. So, `T` is inferred as `String`.
// 4. The lambda parameter `s` is inferred as `String`.
// 5. The lambda has no explicit return, which matches `void`.

// Explicit type declaration (optional, but sometimes needed for clarity or
when inference struggles)
Calculator multiply = (int val1, int val2) -> val1 * val2;
// Here, types `int val1, int val2` are explicitly stated, but often not
necessary.
```

The compiler's ability to infer types is a key feature that makes lambda expressions so concise and powerful in Java.

22. What is a functional interface? Which functional interfaces are predefined and where they are used?

- **What is a Functional Interface?**
  - **Definition:** A functional interface is an interface that contains **exactly one abstract method**.
  - **Purpose:** It acts as a contract for lambda expressions and method references. A lambda expression can only be used where a functional interface is expected.
  - **Annotation:** You can optionally annotate a functional interface with `@FunctionalInterface`. This annotation is a compile-time check to ensure that the interface truly has only one abstract method. It does not affect runtime behavior.
  - **Inheritance:** Default methods, static methods, and methods from `java.lang.Object` (like `equals`, `hashCode`, `toString`) do not count towards the "single abstract method" rule. So, a functional interface can have any number of default or static methods, plus one abstract method.
  - **Example:**

```
@FunctionalInterface // Optional annotation
interface MyConverter {
    int convert(String s); // Single abstract method
    // default void log(String s) { System.out.println(s); } // OK,
default method
    // static void helper() { /* ... */ } // OK, static method
}
```

- **Predefined Functional Interfaces (from `java.util.function` package):** Java 8 introduced a rich set of predefined functional interfaces to cover common use cases. These are categorized based on their parameter and return types:

    1. **`Predicate<T>`:**
        - **Abstract Method:** `boolean test(T t)`
        - **Purpose:** Represents a boolean-valued function of one argument. Used for filtering.
        - **Where used:** `Stream.filter()`, `Collection.removeIf()`.
        - **Example:** `numbers.stream().filter(n -> n % 2 == 0);` (Lambda implements `Predicate<Integer>`)
    2. **`Consumer<T>`:**
        - **Abstract Method:** `void accept(T t)`
        - **Purpose:** Represents an operation that accepts a single input argument and returns no result. Used for performing an action on each element.
        - **Where used:** `Stream.forEach()`, `Iterable.forEach()`.
        - **Example:** `list.forEach(item -> System.out.println(item));` (Lambda implements `Consumer<String>`)
    3. **`Function<T, R>`:**
        - **Abstract Method:** `R apply(T t)`
        - **Purpose:** Represents a function that accepts one argument and produces a result. Used for transformation (mapping).
        - **Where used:** `Stream.map()`.
        - **Example:** `numbers.stream().map(n -> n * n);` (Lambda implements `Function<Integer, Integer>`)
    4. **`Supplier<T>`:**
        - **Abstract Method:** `T get()`
        - **Purpose:** Represents a supplier of results. It takes no arguments and returns a result. Used for lazy initialization or deferred execution.
        - **Where used:** `Optional.orElseGet()`, `Stream.generate()`.
        - **Example:** `Optional.empty().orElseGet(() -> "Default Value");` (Lambda implements `Supplier<String>`)
    5. **`UnaryOperator<T>`:**
        - **Abstract Method:** `T apply(T t)`
        - **Purpose:** Represents an operation on a single operand that produces a result of the same type as its operand. It extends `Function<T, T>`.
        - **Where used:** `Stream.reduce()` (in some forms).
        - **Example:** `list.replaceAll(s -> s.toUpperCase());` (Lambda implements `UnaryOperator<String>`)
    6. **`BinaryOperator<T>`:**

- **Abstract Method:** `T apply(T t1, T t2)`
- **Purpose:** Represents an operation upon two operands of the same type, producing a result of the same type as the operands. It extends `BiFunction<T, T, T>`.
- **Where used:** `Stream.reduce()`, `Stream.collect()`.
- **Example:** `numbers.stream().reduce(0, (a, b) -> a + b);` (Lambda implements `BinaryOperator<Integer>`)

Variations: For primitive types, there are specialized versions (e.g., `IntPredicate`, `DoubleConsumer`, `LongFunction`, `IntSupplier`, `DoubleUnaryOperator`, `LongBinaryOperator`) to avoid auto-boxing/unboxing overhead. There are also `BiPredicate`, `BiConsumer`, `BiFunction` for operations with two arguments.

Advanced Java

1. What all servers/servlet containers are you aware of? Do you know about Jetty?

- **Servers/Servlet Containers I am aware of:**

1. **Apache Tomcat:** The most popular open-source servlet container. It's lightweight, widely used for deploying Java web applications (servlets, JSPs, WebSockets). It implements Java Servlet, JavaServer Pages (JSP), Java Expression Language (EL), and Java WebSocket specifications.
2. **Eclipse Jetty:** Another open-source, lightweight, and highly embeddable HTTP server and Servlet container.
3. **WildFly (formerly JBoss AS):** A full-fledged open-source application server from JBoss Community (Red Hat). It provides a complete Java EE (now Jakarta EE) runtime, including EJB, JMS, JPA, etc.
4. **GlassFish:** An open-source application server developed by Oracle (now managed by Eclipse Foundation) that also implements Jakarta EE.
5. **IBM WebSphere Application Server:** A commercial, enterprise-grade application server.
6. **Oracle WebLogic Server:** Another commercial, enterprise-grade application server.
7. **Undertow:** A flexible, high-performance, non-blocking web server written in Java, primarily used as the default embedded web server for WildFly and Spring Boot.

- **About Jetty:**

  - **Lightweight and Embeddable:** Jetty is known for its small footprint and its ability to be easily embedded directly into Java applications. This makes it ideal for microservices, command-line tools, and device applications where a full-blown application server is overkill.
  - **Asynchronous and Non-Blocking:** Jetty is built on a non-blocking I/O architecture, making it highly efficient for handling a large number of concurrent connections, especially in asynchronous programming models.
  - **Modular:** It's very modular, allowing developers to include only the necessary components.
  - **Protocols:** Supports HTTP/1.1, HTTP/2, WebSockets, and SPDY.
  - **Use Cases:** Common in embedded systems, cloud-based microservices (e.g., Spring Boot can embed Jetty), and for testing purposes due to its fast startup time. It's often chosen when a fast, low-overhead server is needed, and the full Java EE stack is not required.

2. What is REST API? Explain in detail. What is @RestController?

- **REST API (Representational State Transfer Application Programming Interface):**
    - **Definition:** REST is an architectural style for designing networked applications. REST APIs are a way to allow different systems (clients and servers) to communicate with each other over the internet using standard HTTP methods. It's not a protocol or a standard itself, but a set of architectural constraints.
    - **Core Principles (REST Constraints):**
        1. **Client-Server:** Separation of concerns between client (UI logic) and server (data storage/business logic). This improves portability and scalability.
        2. **Stateless:** Each request from client to server must contain all the information needed to understand the request. The server should not store any client context between requests. This improves scalability, reliability, and visibility.
        3. **Cacheable:** Responses from the server should explicitly or implicitly define themselves as cacheable or non-cacheable to prevent clients from reusing stale or inappropriate data.
        4. **Uniform Interface:** This is the most important constraint, simplifying the overall system architecture. It includes:
            - **Resource Identification in Requests:** Resources are identified by URIs (e.g., `/users/123`).
            - **Resource Manipulation Through Representations:** Clients interact with resources by manipulating representations (e.g., JSON, XML) of those resources.
            - **Self-descriptive Messages:** Each message contains enough information to describe how to process the message.
            - **Hypermedia as the Engine of Application State (HATEOAS):** Resources should contain links to other related resources, guiding the client through the application's state transitions. (Often optional in practical REST APIs).
        5. **Layered System:** The client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary. This allows for scalability and security.
        6. **Code on Demand (Optional):** Servers can temporarily extend or customize client functionality by transferring executable code (e.g., JavaScript).
    - **How it Works:**
        - **Resources:** Everything is a resource (e.g., a user, a product, an order). Resources are identified by URLs (URIs).
        - **HTTP Methods:** Standard HTTP methods (verbs) are used to perform operations on these resources:
            - **GET:** Retrieve a resource (safe, idempotent, cacheable).
            - **POST:** Create a new resource or submit data (not idempotent, not cacheable).
            - **PUT:** Update an existing resource (idempotent - multiple identical requests have same effect as one).
            - **DELETE:** Remove a resource (idempotent).
            - **PATCH:** Partially update a resource (not necessarily idempotent).
        - **Representations:** Data is exchanged in representations (e.g., JSON, XML, plain text). JSON is most common today.
        - **Statelessness:** No session information on the server. Each request carries all context.

    - **Standard Status Codes:** HTTP status codes (200 OK, 201 Created, 404 Not Found, 500 Internal Server Error, etc.) indicate the outcome of the request.

- `@RestController` **in Spring Boot:**
    - **Purpose:** `@RestController` is a convenience annotation in Spring Framework, specifically designed for building RESTful web services. It's a stereotype annotation that combines `@Controller` and `@ResponseBody`.
    - `@Controller`**:** Marks a class as a Spring MVC controller, capable of handling incoming web requests.
    - `@ResponseBody`**:** When this annotation is present on a method (or implicitly via `@RestController` on the class), it tells Spring to automatically bind the return value of the method to the web response body. Spring uses HTTP Message Converters (like Jackson for JSON) to convert the Java object return type into a suitable format (e.g., JSON or XML) and write it directly to the HTTP response body. This eliminates the need to explicitly annotate each method with `@ResponseBody`.
    - **Advantages of** `@RestController`**:** Simplifies REST API development by reducing boilerplate code. You no longer need to add `@ResponseBody` to every handler method if you want to return data directly.
    - **Example:**

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;

@RestController // This implies @Controller and @ResponseBody
public class MyRestApi {

    @GetMapping("/hello")
    public String sayHello() {
        return "Hello from Spring Boot REST!"; // String is directly
written to response body
    }

    @GetMapping("/users/{id}")
    public User getUser(@PathVariable Long id) {
        // In a real app, you'd fetch from a service/DB
        return new User(id, "John Doe"); // User object is
automatically converted to JSON
    }
}

class User { // Simple POJO for demonstration
    private Long id;
    private String name;
    // Getters, Setters, Constructor
    public User(Long id, String name) { this.id = id; this.name = name;
```

```
}
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

3. Write a REST program from scratch in SpringBoot.

Here's a basic Spring Boot REST API example for managing `Product` resources.

Project Setup (Conceptual - `pom.xml`):

You'd typically use Spring Initializr (start.spring.io) to generate a project with these dependencies:

- `spring-boot-starter-web`: For building web, including RESTful, applications.
- `spring-boot-starter-data-jpa`: For JPA and database interaction.
- `h2` (or any database driver): In-memory database for simplicity.
- `lombok` (optional, for less boilerplate code).

Code Structure:

```
src/main/java/
└── com/example/demo/
    ├── DemoApplication.java        // Spring Boot entry point
    ├── Product.java                // Entity/Model
    ├── ProductRepository.java      // JPA Repository
    └── ProductController.java      // REST Controller
```

1. `pom.xml` (Excerpt - assuming Maven project):

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>3.2.0</version> <!-- Use a recent version -->
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <groupId>com.example</groupId>
    <artifactId>demo</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>demo</name>
    <description>Demo project for Spring Boot REST API</description>
```

```
    <properties>
        <java.version>17</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <!-- For testing -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <configuration>
                    <excludes>
                        <exclude>
                            <groupId>org.projectlombok</groupId>
                            <artifactId>lombok</artifactId>
                        </exclude>
                    </excludes>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

2. `Product.java` **(Entity/Model):**

```java
package com.example.demo;

import jakarta.persistence.Entity; // For JPA entities
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import lombok.Data; // Lombok for getters/setters/constructors (optional)
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;

@Entity // Marks this class as a JPA entity, mapped to a database table
@Data // Lombok: Generates getters, setters, toString, equals, hashCode
@NoArgsConstructor // Lombok: Generates a no-argument constructor
@AllArgsConstructor // Lombok: Generates a constructor with all fields
public class Product {

    @Id // Marks this field as the primary key
    @GeneratedValue(strategy = GenerationType.IDENTITY) // Auto-incrementing ID
    private Long id;
    private String name;
    private double price;
    private String description;

    // If not using Lombok, you'd manually write:
    // public Product() {}
    // public Product(Long id, String name, double price, String description) {
... }
    // public Long getId() { ... }
    // public void setId(Long id) { ... }
    // ... etc.
}
```

3. `ProductRepository.java` **(Data Access Layer):**

```java
package com.example.demo;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository // Marks this as a Spring Data JPA repository
public interface ProductRepository extends JpaRepository<Product, Long> {
    // JpaRepository provides CRUD methods (save, findById, findAll, deleteById,
etc.)
    // You can add custom query methods here, e.g.:
    List<Product> findByNameContainingIgnoreCase(String name);
}
```

4. `ProductController.java` **(REST Controller):**

```
package com.example.demo;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;
import java.util.Optional;

@RestController // Combines @Controller and @ResponseBody
@RequestMapping("/api/products") // Base URL for all endpoints in this controller
public class ProductController {

    @Autowired // Inject ProductRepository dependency
    private ProductRepository productRepository;

    // GET all products
    // Endpoint: GET /api/products
    @GetMapping
    public List<Product> getAllProducts() {
        return productRepository.findAll();
    }

    // GET product by ID
    // Endpoint: GET /api/products/{id}
    @GetMapping("/{id}")
    public ResponseEntity<Product> getProductById(@PathVariable Long id) {
        Optional<Product> product = productRepository.findById(id);
        return product.map(ResponseEntity::ok) // If product found, return 200 OK
with product
                      .orElseGet(() -> ResponseEntity.notFound().build()); //
Else, return 404 Not Found
    }

    // CREATE a new product
    // Endpoint: POST /api/products
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED) // Return 201 Created on successful
creation
    public Product createProduct(@RequestBody Product product) {
        return productRepository.save(product);
    }

    // UPDATE an existing product
    // Endpoint: PUT /api/products/{id}
    @PutMapping("/{id}")
    public ResponseEntity<Product> updateProduct(@PathVariable Long id,
@RequestBody Product productDetails) {
        Optional<Product> optionalProduct = productRepository.findById(id);
        if (optionalProduct.isPresent()) {
            Product existingProduct = optionalProduct.get();
```

```
            existingProduct.setName(productDetails.getName());
            existingProduct.setPrice(productDetails.getPrice());
            existingProduct.setDescription(productDetails.getDescription());
            Product updatedProduct = productRepository.save(existingProduct);
            return ResponseEntity.ok(updatedProduct);
        } else {
            return ResponseEntity.notFound().build();
        }
    }

    // DELETE a product
    // Endpoint: DELETE /api/products/{id}
    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT) // Return 204 No Content on successful
deletion
    public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
        if (productRepository.existsById(id)) {
            productRepository.deleteById(id);
            return ResponseEntity.noContent().build(); // 204 No Content
        } else {
            return ResponseEntity.notFound().build(); // 404 Not Found
        }
    }
}
```

5. `DemoApplication.java` **(Main Spring Boot Application):**

```
package com.example.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication // Combines @Configuration, @EnableAutoConfiguration,
@ComponentScan
public class DemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

To Run:

1. Save these files in the appropriate folder structure.
2. Make sure you have Maven and Java installed.
3. Open a terminal in the project's root directory (where `pom.xml` is located).
4. Run: `mvn spring-boot:run`
5. The application will start on port 8080 (by default).

Test with cURL or Postman:

- **GET all:** `curl http://localhost:8080/api/products`
- **POST (create):** `curl -X POST -H "Content-Type: application/json" -d "{\"name\":\"Laptop\",\"price\":1200.0,\"description\":\"Powerful notebook\"}" http://localhost:8080/api/products`
- **GET by ID:** `curl http://localhost:8080/api/products/1` (assuming ID 1 was created)
- **PUT (update):** `curl -X PUT -H "Content-Type: application/json" -d "{\"name\":\"Laptop Pro\",\"price\":1300.0,\"description\":\"Updated powerful notebook\"}" http://localhost:8080/api/products/1`
- **DELETE:** `curl -X DELETE http://localhost:8080/api/products/1`

This provides a complete, runnable Spring Boot REST API for basic CRUD operations.

4. What is the significance of `@CrossOrigin`? How does it work?

- **Significance of `@CrossOrigin`:** The `@CrossOrigin` annotation in Spring Framework is used to handle **Cross-Origin Resource Sharing (CORS)**. Its primary significance is to allow or restrict web browsers from making requests to a different domain than the one from which the web page was served. Without it, standard browser security policies (like the Same-Origin Policy) would block such cross-origin requests.

  Why it's needed:

  - **Same-Origin Policy (SOP):** A fundamental security mechanism implemented by web browsers. It prevents a web page from making requests to a different domain than the one that served the web page. For example, a JavaScript running on `http://frontend.com` cannot directly make an AJAX request to `http://backend.com` unless the `backend.com` explicitly allows it.
  - **Microservices/Separate Front-end & Back-end:** In modern web development, it's common to have a front-end application (e.g., React, Angular, Vue) served from one domain/port and a backend REST API served from another. CORS is essential to enable these separate components to communicate.

- **How `@CrossOrigin` works (CORS Mechanism):** CORS works by adding specific HTTP headers to the server's response that tell the browser whether it's allowed to access the resource from a different origin.

  1. **Simple Requests:**
     - For "simple" requests (GET, HEAD, POST with specific `Content-Type`s like `application/x-www-form-urlencoded`, `multipart/form-data`, `text/plain`), the browser sends the actual request with an `Origin` header (e.g., `Origin: http://frontend.com`).
     - If the server's response includes an `Access-Control-Allow-Origin` header matching the client's origin (or `*` for any origin), the browser allows the response to be processed by the JavaScript. Otherwise, it blocks the response.
  2. **Preflight Requests (for non-simple requests):**
     - For "non-simple" requests (e.g., PUT, DELETE, PATCH, or POST with `Content-Type: application/json`, or requests with custom headers), the browser first sends an `OPTIONS` request (the "preflight" request) to the server *before* sending the actual request.

- This preflight request includes headers like `Access-Control-Request-Method` (e.g., `PUT`) and `Access-Control-Request-Headers` (e.g., `Authorization`).
- The server responds to the `OPTIONS` request with headers like `Access-Control-Allow-Origin`, `Access-Control-Allow-Methods`, `Access-Control-Allow-Headers`, and `Access-Control-Max-Age`.
- If the preflight response indicates that the actual request is allowed, the browser then proceeds to send the actual request. If not, the browser immediately blocks the request.

`@CrossOrigin` in Spring Boot: When you apply `@CrossOrigin` in Spring (e.g., at controller level or method level):

- Spring automatically intercepts the requests.
- For incoming `OPTIONS` requests (preflight), it generates the appropriate `Access-Control-Allow-Methods`, `Access-Control-Allow-Headers`, etc., headers in the response based on the annotation's configuration.
- For actual requests, it adds the `Access-Control-Allow-Origin` header to the response.

Usage Example:

```
import org.springframework.web.bind.annotation.*;

@RestController
@RequestMapping("/api/data")
// Allows requests from any origin to any endpoint in this controller
// @CrossOrigin // (default allows all origins)
public class DataController {

    // Allows requests from "http://example.com" and "http://another.com"
    // for this specific method
    @CrossOrigin(origins = {"http://example.com", "http://another.com"},
methods = {RequestMethod.GET, RequestMethod.POST})
    @GetMapping("/public")
    public String getPublicData() {
        return "This is public data.";
    }

    // Allows requests from "http://localhost:3000" (common for
React/Angular dev servers)
    @CrossOrigin(origins = "http://localhost:3000")
    @PostMapping("/secure")
    public String postSecureData(@RequestBody String data) {
        return "Received secure data: " + data;
    }

    // You can also configure it globally in Spring Boot (e.g.,
WebMvcConfigurer)
    // This is often preferred for more complex CORS policies.
}
```

By using `@CrossOrigin`, developers can explicitly tell browsers that it's safe to allow requests from specific origins, thus overcoming the Same-Origin Policy for legitimate cross-domain communication.

5. How do `@ResponseBody` and `@RequestBody` work? What is `ResponseEntity`?

These three components are fundamental in Spring MVC/REST for handling the request and response bodies of HTTP messages.

- **`@RequestBody`:**
  - **Purpose:** This annotation is used on a method parameter to bind the incoming HTTP request body to that parameter. Spring automatically deserializes the content of the request body (e.g., JSON or XML) into an instance of the Java type declared for the parameter.
  - **How it works:**
    1. When a request arrives, Spring MVC determines the content type of the request body (e.g., `Content-Type: application/json`).
    2. It then uses appropriate `HttpMessageConverter` (e.g., `MappingJackson2HttpMessageConverter` for JSON, `Jaxb2HttpMessageConverter` for XML) to convert the raw HTTP request body into the Java object specified by the method parameter.
    3. The converted Java object is then passed as an argument to the controller method.
  - **Example:**

```
@PostMapping("/products")
public Product createProduct(@RequestBody Product product) {
    // The JSON or XML body of the HTTP POST request will be
    // automatically converted into a Product object.
    return productService.save(product);
}
```

    In this example, an incoming JSON `{"name": "New Phone", "price": 999.99}` would be mapped to a `Product` object.

- **`@ResponseBody`:**
  - **Purpose:** This annotation is used on a method or a return type to indicate that the return value of the method should be bound directly to the HTTP response body. Spring automatically serializes the Java object returned by the method into a format (e.g., JSON or XML) and writes it directly to the response.
  - **How it works:**
    1. After the controller method executes and returns a Java object, Spring MVC determines the `Accept` header in the incoming request (e.g., `Accept: application/json`).
    2. It then uses an appropriate `HttpMessageConverter` (the same ones used by `@RequestBody` but in reverse) to convert the Java object into the requested format.
    3. The converted data is then written directly to the HTTP response output stream.
  - **Example:**

```
@GetMapping("/products/{id}")
@ResponseBody // Explicitly states return value should be response body
public Product getProduct(@PathVariable Long id) {
    // The Product object returned will be automatically converted
    // (e.g., to JSON) and sent as the HTTP response body.
    return productService.findById(id);
}
```

Note: The `@RestController` annotation is a convenience annotation that combines `@Controller` and `@ResponseBody` at the class level, meaning all methods within a `@RestController` annotated class implicitly have `@ResponseBody` applied.

- `ResponseEntity`:
  - **Purpose:** `ResponseEntity` is a class that represents the entire HTTP response: status code, headers, and body. It allows developers to have complete control over the HTTP response that is sent back to the client.
  - **How it works:** Instead of just returning a domain object or a String, you wrap your data, desired HTTP status, and any custom headers within a `ResponseEntity` object. Spring then takes this object and constructs the full HTTP response.
  - **Advantages:**
    - **Custom Status Codes:** Return specific HTTP status codes (e.g., `201 Created`, `204 No Content`, `404 Not Found`, `400 Bad Request`) beyond the default `200 OK`.
    - **Custom Headers:** Add specific HTTP headers (e.g., `Location` for `201 Created`, `Cache-Control`).
    - **No Body:** Send a response with just a status code and headers, without a body (e.g., for DELETE operations returning `204 No Content`).
    - **Flexibility:** Provides fine-grained control over the HTTP response in various scenarios.
  - **Example:**

```
@PostMapping("/products")
public ResponseEntity<Product> createProduct(@RequestBody Product
product) {
Product savedProduct = productService.save(product);
// Return 201 Created status, the saved product in the body,
// and a Location header pointing to the new resource.
return ResponseEntity.status(HttpStatus.CREATED)
.body(savedProduct);
}

    @GetMapping("/products/{id}")
    public ResponseEntity<Product> getProduct(@PathVariable Long id) {
        Product product = productService.findById(id);
        if (product != null) {
            return ResponseEntity.ok(product); // Returns 200 OK with
```

```
product in body
        } else {
            return ResponseEntity.notFound().build(); // Returns 404
Not Found
        }
    }

    @DeleteMapping("/products/{id}")
    public ResponseEntity<Void> deleteProduct(@PathVariable Long id) {
        productService.delete(id);
        return ResponseEntity.noContent().build(); // Returns 204 No
Content
    }
    ```

In summary, `@RequestBody` handles incoming data, `@ResponseBody`
handles outgoing data, and `ResponseEntity` gives you complete control
over the entire HTTP response.
```

**6. What is the difference between DTO and Entities? What are the advantages of using DTO? How have you implemented DTOs in your project?**

- **Difference between DTO (Data Transfer Object) and Entities (JPA Entities):**

| Feature | Entity (JPA Entity) | DTO (Data Transfer Object) |
|---|---|---|
| **Purpose** | Represents a table in the database. Core business object. | Transfers data between layers (e.g., API to UI, Service to Controller). |
| **Persistence** | Directly mapped to database tables/columns (ORM). | No direct database mapping. Simple data holder. |
| **Contents** | Often contains all fields corresponding to DB table columns, possibly relationships. May contain business logic. | Contains only the fields necessary for a specific data transfer scenario. Rarely contains business logic. |
| **Usage** | Used within the persistence/data access layer and service layer. | Used at the boundaries of layers, especially between Controller and Client (UI). |
| **Lifecycle** | Managed by JPA/Hibernate (managed, detached, etc.). | Simple POJO, no lifecycle management. |
| **Relationships** | Represents complex relationships (OneToMany, ManyToOne) that can lead to N+1 queries. | Simplifies relationships by flattening or selecting specific fields needed for transfer. |
| **Example** | `User` entity with `id`, `username`, `password`, `roles`, `createdAt`. | `UserDTO` with `id`, `username`, `email` (no password or sensitive fields). |

- **Advantages of Using DTOs:**
```

1. **Reduced Data Transfer:** You can transfer only the necessary data fields between layers or to the client, avoiding over-fetching (sending too much data) or under-fetching (not sending enough for a specific view).
2. **Information Hiding/Security:** You can expose only specific data fields to the client, hiding sensitive information (like passwords, internal IDs, or audit fields) that reside in the entity.
3. **Decoupling Layers:** DTOs decouple your API/presentation layer from your internal domain/persistence model. Changes in the database schema or entity structure don't necessarily force changes in your API contract, and vice-versa.
4. **Simplified API Contract:** APIs often have different data requirements than the underlying database. DTOs allow you to design a clear, concise API contract tailored for specific use cases.
5. **Avoid Lazy Initialization Issues:** When returning JPA entities directly from controllers, lazy-loaded collections can cause `LazyInitializationException` if the session is closed before the data is accessed during JSON serialization. DTOs resolve this by explicitly fetching and mapping only the required data.
6. **Better Performance:** Less data transferred over the wire means faster network operations.

- **How I have implemented DTOs in my project:**

  (Tailor this to your actual project experience, here's a generic example)

  "In my project, we extensively used DTOs to create clear boundaries between our Spring Boot REST API layer and our JPA entity model, especially for client-facing APIs.

  Implementation Steps:

  1. **Define DTO Classes:** For each significant entity, we created one or more corresponding DTO classes (e.g., `User`, `UserRequestDTO`, `UserResponseDTO`, `UserDetailDTO`). These DTOs contained only the fields relevant to the specific API endpoint's input or output. For instance, a `UserRequestDTO` for creating a user might only contain `username` and `password`, while a `UserResponseDTO` might contain `id`, `username`, and `email` but *not* the password.

  2. **Mapping between DTO and Entity:** We used **ModelMapper** (or MapStruct, or manual conversion for complex cases) to map data between DTOs and entities.

     - **In Controller:**
       - When a `POST` or `PUT` request came in, the `@RequestBody` would bind the incoming JSON/XML directly to a DTO (e.g., `UserRequestDTO`).
       - We would then use `ModelMapper` in the **service layer** to convert this `UserRequestDTO` into a `User` entity before saving it to the database.
       - After retrieving data from the repository (an `Entity`), we would map the `Entity` to a `UserResponseDTO` before returning it from the controller.

  Example Code (Conceptual):

```
// 1. User Entity
@Entity
public class User {
    @Id @GeneratedValue private Long id;
    private String username;
```

```
    private String password; // Sensitive
    private String email;
    private String role;     // Internal
    // getters, setters, constructors
}

// 2. User Request DTO (for creating/updating)
@Data @NoArgsConstructor @AllArgsConstructor
public class UserRequestDTO {
    private String username;
    private String password;
    private String email;
}

// 3. User Response DTO (for sending to client)
@Data @NoArgsConstructor @AllArgsConstructor
public class UserResponseDTO {
    private Long id;
    private String username;
    private String email;
}

// 4. In Service Layer (using ModelMapper)
@Service
public class UserService {
    @Autowired private UserRepository userRepository;
    @Autowired private ModelMapper modelMapper; // Configured Spring bean

    public UserResponseDTO createUser(UserRequestDTO userRequestDTO) {
        User user = modelMapper.map(userRequestDTO, User.class); // DTO to
Entity
        user.setRole("USER"); // Set internal fields
        User savedUser = userRepository.save(user);
        return modelMapper.map(savedUser, UserResponseDTO.class); // Entity
to DTO
    }

    public List<UserResponseDTO> getAllUsers() {
        return userRepository.findAll().stream()
            .map(user -> modelMapper.map(user, UserResponseDTO.class))
            .collect(Collectors.toList());
    }
}

// 5. In Controller
@RestController
@RequestMapping("/users")
public class UserController {
    @Autowired private UserService userService;

    @PostMapping
    public ResponseEntity<UserResponseDTO> createUser(@RequestBody
UserRequestDTO userRequestDTO) {
        UserResponseDTO createdUser =
```

```
    userService.createUser(userRequestDTO);
            return new ResponseEntity<>(createdUser, HttpStatus.CREATED);
        }

        @GetMapping
        public List<UserResponseDTO> getAllUsers() {
            return userService.getAllUsers();
        }
    }
```

This approach allowed us to present a clean, consistent API to consumers, hide internal complexities and sensitive data, and provide flexibility for evolving both the database schema and the API independently."

7. How to secure Spring REST API? How authentication and authorization works?

Securing a Spring REST API typically involves **authentication** (who are you?) and **authorization** (what are you allowed to do?). Spring Security is the de-facto standard framework for this in Spring Boot applications.

- **How to Secure Spring REST API (using Spring Security):**

  1. **Add Spring Security Dependency:**

     ```
     <dependency>
         <groupId>org.springframework.boot</groupId>
         <artifactId>spring-boot-starter-security</artifactId>
     </dependency>
     ```

  2. **Configure Security (SecurityFilterChain):** Create a configuration class that extends `WebSecurityConfigurerAdapter` (Spring Security 5.x) or defines a `SecurityFilterChain` bean (Spring Security 6+). Here's a modern Spring Security 6+ example:

     ```
     import org.springframework.context.annotation.Bean;
     import org.springframework.context.annotation.Configuration;
     import
     org.springframework.security.config.annotation.web.builders.HttpSecurit
     y;
     import
     org.springframework.security.config.annotation.web.configuration.Enable
     WebSecurity;
     import org.springframework.security.config.http.SessionCreationPolicy;
     import org.springframework.security.core.userdetails.User;
     import org.springframework.security.core.userdetails.UserDetails;
     import
     org.springframework.security.core.userdetails.UserDetailsService;
     import
     org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
     import org.springframework.security.crypto.password.PasswordEncoder;
     import
     ```

```
org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;

@Configuration
@EnableWebSecurity // Enables Spring Security's web security features
public class SecurityConfig {

    // 1. Password Encoder (crucial for securely storing passwords)
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    // 2. UserDetailsService (how to load user details - e.g., from DB,
in-memory)
    @Bean
    public UserDetailsService userDetailsService(PasswordEncoder
passwordEncoder) {
        // In-memory user for demonstration (for real apps, use
JDBC/JPA UserDetailsService)
        UserDetails user = User.withUsername("user")
            .password(passwordEncoder.encode("password"))
            .roles("USER")
            .build();
        UserDetails admin = User.withUsername("admin")
            .password(passwordEncoder.encode("adminpass"))
            .roles("ADMIN", "USER")
            .build();
        return new InMemoryUserDetailsManager(user, admin);
    }

    // 3. Security Filter Chain (defines authorization rules, session
management, etc.)
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http)
throws Exception {
        http
            .csrf(csrf -> csrf.disable()) // Disable CSRF for REST APIs
(stateless)
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/public/**").permitAll() // Publicly
accessible endpoints
                .requestMatchers("/api/admin/**").hasRole("ADMIN") //
Only ADMIN role can access
                .requestMatchers("/api/**").hasAnyRole("USER", "ADMIN")
// USER or ADMIN can access
                .anyRequest().authenticated() // All other requests
require authentication
            )
            .httpBasic(httpBasic -> {}) // Enable HTTP Basic
authentication
            .sessionManagement(session ->
session.sessionCreationPolicy(SessionCreationPolicy.STATELESS)); //
REST APIs are typically stateless
```

```
            return http.build();
        }
    }
```

3. **Authentication Mechanisms:**
    - **HTTP Basic Authentication:** Simple username/password sent in an `Authorization` header. (Easy to implement, less secure over plain HTTP).
    - **Form-based Authentication:** (More common for traditional web apps, less so for pure REST APIs unless you build a login page).
    - **Token-based Authentication (JWT - JSON Web Tokens):** Most common for REST APIs. Client logs in once, gets a JWT, sends it in `Authorization: Bearer <token>` header for subsequent requests. Server validates token. (Requires additional libraries like `jjwt`).
    - **OAuth2 / OpenID Connect:** For delegated authorization (e.g., "Login with Google").

- **How Authentication and Authorization Works:**

  1. Authentication:

  - **Definition:** The process of verifying the identity of a user or client. It answers the question: "Who are you?".
  - **Process:**
    1. **Credentials Submission:** A client (e.g., web browser, mobile app) sends credentials (username/password, token, API key) to the server.
    2. **Spring Security Filters:** Spring Security has a chain of filters. One of these (e.g., `BasicAuthenticationFilter`, `UsernamePasswordAuthenticationFilter`, or a custom JWT filter) intercepts the request.
    3. **`AuthenticationManager`:** The filter passes the credentials (wrapped in an `Authentication` object) to the `AuthenticationManager`.
    4. **`AuthenticationProvider`:** The `AuthenticationManager` delegates to one or more `AuthenticationProvider`s.
    5. **`UserDetailsService`:** An `AuthenticationProvider` typically uses a `UserDetailsService` to retrieve user details (username, password, roles) from a data source (database, LDAP, in-memory).
    6. **Password Matching:** The provided password from the client is then compared to the stored password (after being encoded) using a `PasswordEncoder`.
    7. **Success/Failure:**
        - **Success:** If credentials match, an authenticated `Authentication` object (containing user details and authorities/roles) is created and stored in the `SecurityContextHolder`. This context is available throughout the request's lifecycle.
        - **Failure:** If credentials don't match, an `AuthenticationException` is thrown, and Spring Security typically returns an HTTP `401 Unauthorized` status.

  2. Authorization:

- **Definition:** The process of determining if an authenticated user has permission to perform a certain action or access a particular resource. It answers the question: "Are you allowed to do this?".
- **Process:**
    1. **After Authentication:** Once a user is authenticated, their granted authorities (roles/permissions) are stored in the `SecurityContextHolder`.
    2. **`AccessDecisionManager`:** Before a request reaches the controller, Spring Security's `AccessDecisionManager` (part of the authorization process) inspects the requested resource or method.
    3. **`AccessDecisionVoter`s:** The `AccessDecisionManager` delegates to one or more `AccessDecisionVoter`s (e.g., `RoleVoter`, `ExpressionBasedVoter`).
    4. **Security Expressions:** Authorization rules are often defined using Spring Security expressions (e.g., `hasRole('ADMIN')`, `hasAuthority('READ_USER')`, `isAuthenticated()`, `permitAll()`).
    5. **Decision:** The `AccessDecisionVoter`s determine if the authenticated user's authorities match the required permissions for the resource/method.
    6. **Success/Failure:**
        - **Success:** If authorization passes, the request proceeds to the controller.
        - **Failure:** If authorization fails, an `AccessDeniedException` is thrown, and Spring Security typically returns an HTTP `403 Forbidden` status.

In essence: Authentication is about *who you are*, and authorization is about *what you can do*. Spring Security seamlessly integrates these processes through its filter chain and configuration.

8. What is Spring Boot? What do you mean by opinionated defaults? How does auto-configuration work?

- **What is Spring Boot?** Spring Boot is an opinionated framework that simplifies the development of stand-alone, production-ready Spring-based applications. It aims to get Spring applications up and running with minimal configuration and effort.

  Key Goals/Features:

    - **Rapid Application Development:** Significantly reduces development time and boilerplate code.
    - **Convention over Configuration:** Favors sensible defaults, reducing the need for explicit XML or Java configurations.
    - **Embedded Servers:** Allows embedding popular servlet containers like Tomcat, Jetty, or Undertow directly into the application, making it executable as a single JAR file.
    - **Production-Ready Features:** Provides features like health checks, metrics, externalized configuration, and security out-of-the-box via Spring Boot Actuator.
    - **No XML Configuration:** Largely eliminates the need for XML configuration, favoring annotation-based and Java-based configurations.

- **What do you mean by opinionated defaults?** "Opinionated defaults" means that Spring Boot makes certain assumptions and provides sensible, pre-configured settings for common use cases. Instead of requiring developers to explicitly configure every aspect, Spring Boot provides a working setup out of the box based on the dependencies present in the classpath.

    - **Example:**

        - If you include `spring-boot-starter-web` in your project, Spring Boot automatically configures an embedded Tomcat server, Spring MVC, and JSON serialization (Jackson). You don't need to configure Tomcat's port, DispatcherServlet, or MessageConverters unless you want to override the default.
        - If you include `spring-boot-starter-data-jpa` and a database driver like H2, Spring Boot automatically configures an in-memory H2 database, a `DataSource`, and `EntityManagerFactory`.
    - **Advantages:**
        - **Faster Development:** Developers can get started quickly without deep knowledge of all underlying configuration options.
        - **Reduced Boilerplate:** Less code to write and maintain.
        - **Consistency:** Encourages consistent project structures and configurations across different applications.
        - **Easily Overridable:** While opinionated, these defaults are not rigid. Developers can easily override them by providing their own configurations (e.g., in `application.properties`/`application.yml` or custom `@Configuration` classes).

- **How does auto-configuration work?** Spring Boot's auto-configuration is the core mechanism that enables "opinionated defaults" and "convention over configuration."

    1. **`@SpringBootApplication`:** This annotation (or explicitly `@EnableAutoConfiguration`) on your main application class triggers the auto-configuration process.
    2. **Classpath Conditions:** Spring Boot scans the classpath for various libraries and configurations.
    3. **`META-INF/spring/org.springframework.boot.autoconfigure.AutoConfiguration.imports`:** This file (or `spring.factories` in older versions) within Spring Boot's auto-configure JARs (e.g., `spring-boot-autoconfigure.jar`) lists all the auto-configuration classes provided by Spring Boot. Each `*AutoConfiguration` class contains logic for configuring specific features (e.g., `WebMvcAutoConfiguration`, `DataSourceAutoConfiguration`).
    4. **Conditional Logic (`@ConditionalOnClass`, `@ConditionalOnMissingBean`, etc.):** Each auto-configuration class uses Spring's `@Conditional` annotations. These annotations check for specific conditions on the classpath or in the application context before an auto-configuration is applied.
        - `@ConditionalOnClass`: Only apply if a specific class is present on the classpath (e.g., `WebMvcAutoConfiguration` will only apply if `DispatcherServlet` class is found).
        - `@ConditionalOnMissingBean`: Only apply if a bean of a specific type is *not* already defined by the user. This is crucial for "opinionated defaults" – if you define your own `DataSource` bean, Spring Boot won't try to auto-configure one.
        - `@ConditionalOnProperty`: Only apply if a specific property (e.g., `spring.datasource.url`) is set to a certain value.
    5. **Bean Definition:** If all conditions are met, the auto-configuration class defines and registers the necessary Spring beans (e.g., `DispatcherServlet`, `DataSource`, `RestTemplate`, `JdbcTemplate`) into the Spring application context.

    Example: When you add `spring-boot-starter-web` to your `pom.xml`, it pulls in `spring-webmvc.jar` and `tomcat-embed-core.jar`.

- `WebMvcAutoConfiguration` detects `DispatcherServlet` (`@ConditionalOnClass(DispatcherServlet.class)`) and configures Spring MVC.
- `EmbeddedWebServerFactoryCustomizerAutoConfiguration` detects Tomcat (`@ConditionalOnClass(Tomcat.class)`) and configures an embedded Tomcat server.
- If you then define your own `DispatcherServlet` bean, `WebMvcAutoConfiguration`'s `@ConditionalOnMissingBean` will prevent it from registering its default one, respecting your custom configuration.

This intelligent conditional logic is what makes Spring Boot's auto-configuration so powerful and flexible.

9. What is the difference between `@Component` and `@Bean`? What will happen if I annotate an empty class with `@Component`?

- **Difference between `@Component` and `@Bean`:**

Feature	`@Component`	`@Bean`
Location	Placed on a **class**.	Placed on a **method** within a `@Configuration` class.
Role	Indicates that a class is a Spring-managed component (a bean).	Indicates that a method produces a Spring-managed bean.
Instance Creation	Spring container itself creates and manages the instance via component scanning.	You (the developer) explicitly create the instance within the method, and Spring registers it.
Control	Less control over instance creation (Spring handles it).	More control over instance creation (you write the instantiation logic).
Use Case	Your own classes, where you own the source code and want Spring to manage their lifecycle.	Useful for configuring third-party library classes, or when you need complex setup logic for a bean.
Stereotypes	`@Service`, `@Repository`, `@Controller`, `@RestController` are specializations of `@Component`.	
Example	`java @Component class MyService {}`	`java @Configuration class AppConfig { @Bean MyService myService() { return new MyService(); } }`

Analogy:

- `@Component` is like telling Spring: "Hey, this class is important, figure out how to create an instance of it and manage it for me."
- `@Bean` is like telling Spring: "Here's a method. When you call this method, it will return an object that I want *you* to manage as a bean."

- **What will happen if I annotate an empty class with `@Component`?**

If you annotate an empty class with `@Component`, Spring's **component scanning** mechanism will discover this class.

1. **Discovery:** When the Spring application starts, and component scanning is enabled (which it is by default in Spring Boot applications via `@SpringBootApplication`), it will find your `EmptyClass`.
2. **Bean Registration:** Spring will then create an instance of this `EmptyClass` and register it as a **Spring bean** in its Application Context. The bean's name will typically be the lowercase version of the class name (e.g., `emptyClass`).
3. **No Error:** There will be no errors. An empty class is a valid class.
4. **Functionality:** The bean itself won't *do* anything useful on its own, as it has no methods or state. However, it will be available for **dependency injection** into other components.

Example:

```
package com.example.demo;

import org.springframework.stereotype.Component;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@Component
class EmptyClass {
    public void sayHello() {
        System.out.println("Hello from EmptyClass!");
    }
}

@Component
class MyConsumerComponent implements CommandLineRunner {
    @Autowired
    private EmptyClass emptyClass; // Spring will inject the instance of
EmptyClass

    @Override
    public void run(String... args) throws Exception {
        if (emptyClass != null) {
            System.out.println("EmptyClass bean successfully injected.");
            emptyClass.sayHello();
        }
    }
}

@SpringBootApplication
public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
}
```

When this `DemoApplication` runs, you'll see "EmptyClass bean successfully injected." and "Hello from EmptyClass!" in the console, confirming that Spring created and managed an instance of `EmptyClass` even though it was initially empty (I added `sayHello` just to demonstrate interaction).

10. What is IoC and Dependency Injection? What are different types?

- **IoC (Inversion of Control):**
  - **Definition:** IoC is a design principle where the control of object creation, dependency management, and lifecycle management is transferred from the application code to a framework or container (like Spring). Instead of the application creating its own dependencies, the framework creates and provides them.
  - **"Inversion"**: Traditionally, your code would *call* a library or framework. With IoC, the framework *calls back* into your code. It's about who initiates the call. In terms of object dependencies, instead of a component *requesting* its dependencies (and thus controlling their creation), the framework *provides* them (thus inverting the control of dependency resolution).
  - **Analogy:** Instead of you (the component) building your own car (dependencies), the car manufacturer (IoC container) builds and delivers the car to you.
- **Dependency Injection (DI):**
  - **Definition:** DI is a specific implementation of the IoC principle. It's a pattern where the dependencies of an object (the services or objects it needs to perform its task) are "injected" into it, rather than the object creating them itself or looking them up.
  - **How it Works (Spring Context):** The Spring IoC container is responsible for creating objects, configuring them, and wiring their dependencies. When the container creates a bean, it checks for its dependencies and provides them.
  - **Analogy:** The car manufacturer (IoC container) explicitly gives you the engine, wheels, seats, etc., (dependencies) that make up your car.
- **Relationship:** Dependency Injection is a mechanism or pattern for achieving Inversion of Control. IoC is the broader principle, and DI is a concrete way to implement it.
- **Different Types of Dependency Injection:** DI can be performed in several ways, commonly supported by frameworks like Spring:
  1. **Constructor Injection:**
     - **Mechanism:** Dependencies are provided as arguments to the class's constructor. This is generally considered the **most preferred** method because it ensures that all required dependencies are present at the time an object is created, making the object's state valid from its inception. It also promotes immutability if fields are `final`.
     - **Example (Spring):**

```
@Service
public class MyService {
    private final MyRepository repository; // Make it final for
immutability
```

```
    // Dependencies are injected via the constructor
    @Autowired // Optional from Spring 4.3 if only one constructor
    public MyService(MyRepository repository) {
        this.repository = repository;
    }

    public void doBusinessLogic() {
        // use repository
    }
}
```

2. **Setter Injection:**
    - **Mechanism:** Dependencies are provided via public setter methods of the class. The container calls these setter methods after the object has been instantiated.
    - **Advantage:** Allows for optional dependencies (setters might not always be called).
    - **Disadvantage:** The object can be in an invalid state until all setters are called. Doesn't guarantee required dependencies are present. Not suitable for immutable dependencies.
    - **Example (Spring):**

```
@Service
public class MyService {
    private MyRepository repository;

    // Dependencies are injected via a setter method
    @Autowired
    public void setRepository(MyRepository repository) {
        this.repository = repository;
    }

    public void doBusinessLogic() {
        // use repository
    }
}
```

3. **Field Injection (or Property Injection):**
    - **Mechanism:** Dependencies are injected directly into instance variables (fields) using annotations. Spring uses reflection to inject the dependencies after the object has been instantiated.
    - **Advantage:** Very concise, requires minimal code.
    - **Disadvantages:**
        - Breaks encapsulation (private fields are modified externally).

- Harder to test independently (without a Spring container) as you can't manually set dependencies easily.
- Can lead to null pointer exceptions if the field is not injected (e.g., if you instantiate the class yourself with `new`).
- Discouraged by Spring team for required dependencies.

- **Example (Spring):**

```
@Service
public class MyService {
    @Autowired // Dependency injected directly into the field
    private MyRepository repository;

    public void doBusinessLogic() {
        // use repository
    }
}
```

Recommendation: Constructor injection is generally the most recommended approach for required dependencies as it ensures immutability, clearer dependency declarations, and better testability. Field injection is often used for optional dependencies or in legacy code, but it's less preferred for core dependencies. Setter injection is useful for optional dependencies or when you need to reconfigure an object after creation.

11. What is the use of `@Autowired`? How autowiring resolves the field?

- **What is the use of `@Autowired`?** The `@Autowired` annotation in Spring is used for **automatic dependency injection**. It tells the Spring IoC (Inversion of Control) container to automatically resolve and inject collaborative beans (dependencies) into the annotated component (bean). Essentially, it eliminates the need for manual configuration (like XML bean definitions or explicit setter calls) to connect one bean to another.
  - **Location:** Can be used on:
    - **Constructors:** (`@Autowired public MyService(MyRepository repo) { ... }`)
    - **Setter methods:** (`@Autowired public void setRepo(MyRepository repo) { ... }`)
    - **Fields/Properties:** (`@Autowired private MyRepository repo;`)
- **How Autowiring Resolves the Field (or Constructor/Setter Parameter):**

  Spring uses a specific strategy to resolve the dependency indicated by `@Autowired`:

  1. **By Type (Primary Strategy):**
     - Spring first tries to find a single bean in its application context that matches the **type** of the dependency being injected.
     - **Example:** If you have `@Autowired private MyRepository repository;`, Spring will look for a bean of type `MyRepository` (or a subclass/implementation of `MyRepository`).
     - **Scenario 1: Exactly One Matching Bean:** If Spring finds exactly one bean of the required type, it injects that bean.

- **Scenario 2: No Matching Bean:** If Spring finds no bean of the required type, it throws a `NoSuchBeanDefinitionException` (unless `required = false` is set on `@Autowired`).
- **Scenario 3: Multiple Matching Beans (Ambiguity):** If Spring finds *multiple* beans of the same type, it leads to ambiguity and throws a `NoUniqueBeanDefinitionException`. In this case, Spring needs additional hints to know which specific bean to inject.

2. **By Name (When Multiple Matches by Type):**

- If there are multiple beans of the same type, Spring then attempts to resolve the dependency **by name**.
- It looks for a bean whose name matches the name of the field, parameter, or setter method being injected.
- **Example:** If you have `@Autowired private MyRepository impl1Repository;` and there are two `MyRepository` implementations, `Impl1Repository` (bean name `impl1Repository`) and `Impl2Repository` (bean name `impl2Repository`), Spring will try to inject the bean named `impl1Repository`.
- **Note:** The bean name is typically the lowercase, decapitalized version of the class name (e.g., `Impl1Repository` becomes `impl1Repository`). You can also explicitly set a bean name using `@Component("myCustomRepo")` or `@Bean("myCustomRepo")`.

3. **By `@Qualifier` (Explicitly Resolving Ambiguity):**

- When there are multiple beans of the same type and the name-based resolution is not sufficient (e.g., you want to inject a bean with a different name, or you have multiple beans with similar names), you can use the `@Qualifier` annotation.

- `@Qualifier` provides an explicit name to uniquely identify the desired bean.

- **Example:**

```
@Component("primaryRepo")
class PrimaryRepository implements MyRepository { /* ... */ }

@Component("secondaryRepo")
class SecondaryRepository implements MyRepository { /* ... */ }

@Service
public class MyService {
    @Autowired
    @Qualifier("primaryRepo") // Explicitly tell Spring to inject
the bean named "primaryRepo"
    private MyRepository repository;

    // ...
}
```

4. **`@Primary` (Default Candidate):**

- If you have multiple beans of the same type, and one of them is generally preferred, you can annotate that bean with `@Primary`. Spring will then inject this bean by default when ambiguity occurs, unless a specific `@Qualifier` is used.

Resolution Order (Simplified):

1. Find all beans of the required **Type**.
2. If multiple found, check for a matching **Name**.
3. If still ambiguous, check for `@Qualifier` annotation.
4. If still ambiguous and no `@Qualifier`, check for `@Primary` annotation on one of the candidates.
5. If still ambiguous, throw `NoUniqueBeanDefinitionException`.

`@Autowired` greatly simplifies wiring dependencies, making your code cleaner and more testable.

12. What is Maven? Explain role of `pom.xml`. Which dependencies have you used (in your project)?

- **What is Maven? Apache Maven** is a powerful **build automation tool** primarily used for Java projects. It simplifies the entire build lifecycle, including compiling code, running tests, packaging applications (JAR, WAR), and managing project dependencies. It operates based on the concept of a **Project Object Model (POM)**.

  Key features of Maven:

  - **Standardized Project Structure:** Enforces a conventional directory layout, making it easier for new developers to understand any Maven project.
  - **Dependency Management:** Automatically downloads and manages project dependencies (libraries and their transitive dependencies) from remote repositories (like Maven Central).
  - **Build Lifecycle Management:** Defines clear phases (e.g., `compile`, `test`, `package`, `install`, `deploy`) that can be executed sequentially.
  - **Plugin-based Architecture:** Most of Maven's functionalities are provided by plugins, making it extensible.
  - **Consistent Builds:** A project built with Maven will produce consistent results across different environments.

- **Explain role of `pom.xml`:** `pom.xml` (Project Object Model) is the **fundamental configuration file** for Maven projects. It's an XML file that contains all the essential information about the project, its build process, and its dependencies. It acts as a single source of truth for your project.

  Key elements and their roles in `pom.xml`:

  1. **`<modelVersion>`:** Specifies the version of the POM model being used (usually 4.0.0).
  2. **`<groupId>`, `<artifactId>`, `<version>` (GAV Coordinates):** These three elements uniquely identify a project or a dependency in the Maven repository.
     - `groupId`: The unique identifier for your organization or project group (e.g., `com.mycompany.myproject`).
     - `artifactId`: The unique identifier for the project within the group (e.g., `my-application`).
     - `version`: The version of the project (e.g., `1.0.0-SNAPSHOT`, `2.1.3`).
  3. **`<packaging>`:** Defines the type of packaging for the project (e.g., `jar`, `war`, `pom`).
  4. **`<name>`, `<description>`:** Provides human-readable names and descriptions for the project.

5. `<parent>`**:** If a project inherits from a parent POM (common in multi-module projects or Spring Boot projects using a "starter parent"), this section defines the parent's GAV coordinates. This allows inheriting configurations, dependencies, and plugin management.
6. `<properties>`**:** Defines custom properties that can be used throughout the POM, often for managing versions of dependencies or other configuration values (e.g., `<java.version>17</java.version>`).
7. `<dependencies>`**:** This is a crucial section where you declare all the external libraries (dependencies) that your project needs to compile, test, or run. Maven automatically downloads these dependencies and their transitive dependencies.
    - Each dependency is defined by its own `groupId`, `artifactId`, `version`, and optionally `scope` (e.g., `compile`, `test`, `runtime`, `provided`).
8. `<build>`**:** Configures the build process.
    - `<plugins>`**:** Declares plugins that perform specific build tasks (e.g., `maven-compiler-plugin` for compiling, `spring-boot-maven-plugin` for packaging an executable JAR).
9. `<repositories>`**:** Specifies custom remote repositories where Maven should look for dependencies (besides Maven Central).
10. `<profiles>`**:** Defines different build profiles (e.g., for `dev`, `test`, `prod` environments) that can activate different configurations or dependencies.

- **Which dependencies have you used (in your project)?** (This answer should be tailored to *your actual project*. Below is a generic example for a typical Spring Boot REST API with a database.)

  "In my recent Spring Boot REST API project, the primary dependencies I used in `pom.xml` were:

    - `spring-boot-starter-web`: This is the core starter for building web applications using Spring MVC. It includes embedded Tomcat, Spring MVC, and RESTful capabilities (like JSON/XML support via Jackson).
    - `spring-boot-starter-data-jpa`: Essential for interacting with a relational database using Java Persistence API (JPA) and Spring Data JPA. It pulls in Hibernate as the default JPA provider.
    - `h2` **(with** `scope=runtime`**)**: An in-memory database, which I used for local development and testing due to its simplicity and fast startup. For production, this would be replaced with a driver for MySQL, PostgreSQL, Oracle, etc.
    - `lombok` **(with** `optional=true`**)**: A popular library that reduces boilerplate code by automatically generating getters, setters, constructors, `toString()`, `equals()`, and `hashCode()` methods using annotations. This keeps the model classes much cleaner.
    - `spring-boot-starter-security`: For implementing authentication and authorization for the REST endpoints. It provides robust security features.
    - `jjwt-api`, `jjwt-impl`, `jjwt-jackson`: These are libraries for implementing JWT (JSON Web Token) based authentication, which is a common practice for stateless REST APIs.
    - `spring-boot-starter-test` **(with** `scope=test`**)**: Provides comprehensive testing utilities for Spring Boot applications, including Spring Test, JUnit 5, Mockito, and AssertJ.
    - **Database-specific connector (e.g.,** `mysql-connector-java` **or** `postgresql`**):** For connecting to the actual production database."

13. What is SQL injection? How to prevent it in JDBC? Explain with code.

- **What is SQL Injection?** SQL injection is a code injection technique where malicious SQL statements are inserted into an input field (e.g., username, password, search box) by a user to execute predefined SQL

commands. These commands can then manipulate the database, bypass authentication, extract sensitive data, or even destroy data.

Example Scenario: Imagine a login query like: `SELECT * FROM users WHERE username = 'user_input_username' AND password = 'user_input_password';`

If a malicious user enters `admin' OR '1'='1` for `username` and anything for `password`, the query becomes: `SELECT * FROM users WHERE username = 'admin' OR '1'='1' AND password = 'some_password';` Since `'1'='1'` is always true, the `WHERE` clause becomes true, potentially allowing the attacker to log in as 'admin' without knowing the password.

- **How to Prevent it in JDBC?** The most effective and standard way to prevent SQL injection in JDBC is by using **Prepared Statements**.
    - **Prepared Statements:**
        - Prepared Statements pre-compile the SQL query. This means the structure of the query is fixed before any user input is inserted.
        - Placeholders (`?`) are used in the query to represent where user input will go.
        - User input is then set as parameters using `setString()`, `setInt()`, etc., methods.
        - The database distinguishes between the SQL code and the data values, preventing the injected malicious code from being interpreted as part of the SQL command.
- **Explain with Code:**

    1. Vulnerable Code (DO NOT USE):

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class SQLInjectionVulnerable {

    public static void main(String[] args) {
        String username = "admin";
        // Malicious input: ' OR '1'='1 --
        // The "--" comments out the rest of the query (password part)
        String password = "' OR '1'='1 --";

        Connection connection = null;
        Statement statement = null;
        ResultSet resultSet = null;

        try {
            connection = DriverManager.getConnection("jdbc:h2:mem:testdb",
"sa", "");
            statement = connection.createStatement();

            // DANGEROUS: Concatenating user input directly into the SQL
query
```

```
            String sql = "SELECT * FROM users WHERE username = '" + username
+ "' AND password = '" + password + "'";
            System.out.println("Executing SQL (Vulnerable): " + sql);

            resultSet = statement.executeQuery(sql);

            if (resultSet.next()) {
                System.out.println("VULNERABLE: Login Successful! User: " +
resultSet.getString("username"));
            } else {
                System.out.println("Login Failed (Vulnerable).");
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            // Close resources
            try { if (resultSet != null) resultSet.close(); } catch
(SQLException e) {}
            try { if (statement != null) statement.close(); } catch
(SQLException e) {}
            try { if (connection != null) connection.close(); } catch
(SQLException e) {}
        }
    }
}
```

If you run this with the malicious input, it will likely result in a "Login Successful" message, demonstrating the vulnerability.

2. Secure Code (Using Prepared Statements - RECOMMENDED):

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement; // Import PreparedStatement
import java.sql.ResultSet;
import java.sql.SQLException;

public class SQLInjectionPrevention {

    public static void main(String[] args) {
        String username = "admin";
        // Malicious input, but now it's treated as data, not code
        String password = "' OR '1'='1 --";

        Connection connection = null;
        PreparedStatement preparedStatement = null; // Use PreparedStatement
        ResultSet resultSet = null;

        try {
            connection = DriverManager.getConnection("jdbc:h2:mem:testdb",
```

```
"sa", "");

            // Initialize a dummy table for testing
            connection.createStatement().execute("CREATE TABLE users (id INT
PRIMARY KEY AUTO_INCREMENT, username VARCHAR(50), password VARCHAR(50));");
            connection.createStatement().execute("INSERT INTO users
(username, password) VALUES ('legituser', 'legitpass');");
            connection.createStatement().execute("INSERT INTO users
(username, password) VALUES ('admin', 'adminpass');");


            // SAFE: Use placeholders (?) in the SQL query
            String sql = "SELECT * FROM users WHERE username = ? AND
password = ?";
            preparedStatement = connection.prepareStatement(sql);

            // Set user inputs as parameters, binding them safely
            preparedStatement.setString(1, username); // Set first '?'
            preparedStatement.setString(2, password); // Set second '?'

            System.out.println("Executing SQL (Secure): " +
preparedStatement.toString()); // Note: preparedStatement.toString() might
not show the actual parameters substituted.

            resultSet = preparedStatement.executeQuery();

            if (resultSet.next()) {
                System.out.println("SECURE: Login Successful! User: " +
resultSet.getString("username"));
            } else {
                System.out.println("Login Failed (Secure)."); // This will
now correctly fail for malicious input.
            }

        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            // Close resources
            try { if (resultSet != null) resultSet.close(); } catch
(SQLException e) {}
            try { if (preparedStatement != null) preparedStatement.close();
} catch (SQLException e) {}
            try { if (connection != null) connection.close(); } catch
(SQLException e) {}
        }
    }
}
```

In the secure code, when `password` is set, `setString()` correctly escapes the single quotes, treating the entire malicious string as a literal value for the password, rather than part of the SQL command. Thus, the query will correctly fail to log in the attacker.

14. Explain Spring Data architecture. How is it related to Hibernate?

- **Spring Data Architecture:** Spring Data is a powerful umbrella project within the Spring ecosystem that aims to simplify data access for various persistence technologies. It provides a consistent, Spring-based programming model for working with different data stores, be it relational databases, NoSQL databases, or even cloud services.

  Core Components/Architecture:

  1. **Repository Abstraction (`Repository` interface):**
     - This is the foundation. It's a marker interface. All other Spring Data repository interfaces extend it.
     - It doesn't contain any methods but serves as a base to indicate that an interface is a "repository."
  2. **CRUD Repository (`CrudRepository` interface):**
     - Extends `Repository`.
     - Provides basic CRUD (Create, Read, Update, Delete) operations out-of-the-box (e.g., `save()`, `findById()`, `findAll()`, `deleteById()`). You just need to declare an interface extending this, and Spring Data generates the implementation at runtime.
  3. **Paging and Sorting Repository (`PagingAndSortingRepository` interface):**
     - Extends `CrudRepository`.
     - Adds methods for pagination (`findAll(Pageable pageable)`) and sorting (`findAll(Sort sort)`).
  4. **JPA Repository (`JpaRepository` interface - for Spring Data JPA):**
     - Extends `PagingAndSortingRepository`.
     - Specifically tailored for JPA-based data stores. It adds JPA-specific methods like `flush()`, `saveAndFlush()`, `deleteInBatch()`, etc.
     - This is the most commonly used interface for relational databases in Spring Boot.
  5. **Custom Query Methods:**
     - Spring Data's magic lies in its ability to **automatically derive queries from method names**. For example, `findByLastName(String lastName)` will automatically generate the SQL `SELECT * FROM ... WHERE last_name = ?`.
     - It supports a rich set of keywords (`And`, `Or`, `Between`, `Like`, `GreaterThan`, `LessThan`, `OrderBy`, `Containing`, `IgnoreCase`, etc.).
     - For more complex queries, you can use `@Query` annotation (JPQL or native SQL) or implement custom repository methods.
  6. **Behind the Scenes (Specific to Spring Data JPA):**
     - Spring Data JPA takes your repository interfaces and, at runtime, generates concrete implementations using proxies.

    - These generated implementations use the underlying JPA provider (e.g., Hibernate, EclipseLink) to interact with the database.
    - It automatically handles boilerplate code like `EntityManager` management, transaction management, exception translation (from JPA exceptions to Spring's `DataAccessException` hierarchy).

- **How is it related to Hibernate?**

  Hibernate is a JPA Implementation.

  - **JPA (Java Persistence API):** JPA is a **specification** (a standard API) for managing relational data in Java applications. It defines a set of interfaces and annotations for Object-Relational Mapping (ORM).
  - **Hibernate:** Hibernate is a popular, open-source **implementation** of the JPA specification. It's an ORM framework that translates Java objects into relational database tables and vice-versa.

  Relationship with Spring Data JPA: Spring Data JPA sits on top of JPA (and thus typically Hibernate, its most common implementation) to further simplify data access.

  - **Spring Data JPA provides an abstraction layer over JPA.** It doesn't *replace* JPA or Hibernate; rather, it *enhances* and *simplifies* their usage.
  - **Delegation:** When you use `JpaRepository` methods (like `save`, `findById`), Spring Data JPA doesn't perform the database operations itself. Instead, it delegates these operations to the underlying **JPA** `EntityManager`, which in turn is implemented by **Hibernate** (or another JPA provider) to interact with the database.
  - **Reduced Boilerplate:** Spring Data JPA removes the need to write repetitive DAO (Data Access Object) or repository implementations. You define interfaces, and Spring Data handles the actual implementation using Hibernate's capabilities.
  - **Seamless Integration:** Spring Data JPA integrates Hibernate's `EntityManager` with Spring's transaction management and exception handling.

  Analogy:

  - **JPA is the blueprint/contract** for ORM.
  - **Hibernate is the builder** that implements the blueprint.
  - **Spring Data JPA is the architect/manager** that makes it super easy to tell the builder (Hibernate) what to build (your repositories) without writing all the construction details.

  In most Spring Boot applications, when you include `spring-boot-starter-data-jpa`, Hibernate is implicitly included as the default JPA provider. So, while you primarily interact with Spring Data JPA interfaces, Hibernate is silently working underneath to handle the actual ORM tasks.

15. Explain hibernate/JPA entity life cycle?

The JPA (and by extension, Hibernate) entity life cycle describes the different states an entity instance can be in during its lifetime, from being newly created to being managed, detached, and eventually removed. Understanding these states is crucial for effective use of JPA/Hibernate.

There are primarily four states:

1. **New (Transient) State:**

- **Description:** An entity instance is in the `New` or `Transient` state immediately after it has been instantiated using the `new` operator.
- **Characteristics:**
  - It has no persistent identity (no primary key assigned yet, unless manually set).
  - It is not associated with any `EntityManager` (JPA) or `Session` (Hibernate).
  - It has no representation in the database.
- **Transition:**
  - To `Managed` state: By calling `entityManager.persist(entity)` or `session.save(entity)`.
- **Example:** `Product newProduct = new Product("Laptop", 1200.0);`

2. **Managed (Persistent) State:**

- **Description:** An entity instance is in the `Managed` or `Persistent` state when it is associated with an active `EntityManager`/`Session` and its changes are tracked.
- **Characteristics:**
  - It represents a row in the database.
  - Changes made to a managed entity's fields are automatically detected and synchronized with the database when the transaction commits (or `flush()` is called). This is called "dirty checking."
  - It has a persistent identity (its primary key is managed).
- **Transition:**
  - From `New`: By `persist()`, `save()`.
  - From `Detached`: By `merge()`, `update()`.
  - To `Detached` state: When the `EntityManager`/`Session` is closed, cleared, or the transaction commits (and the entity is no longer referenced in the current persistence context).
  - To `Removed` state: By calling `entityManager.remove(entity)`.
- **Example:**

```
Product managedProduct = new Product("Desktop", 800.0);
entityManager.persist(managedProduct); // managedProduct is now in
Managed state
managedProduct.setPrice(750.0);       // Change is tracked
// Transaction commit will persist/update changes
```

3. **Detached State:**

- **Description:** An entity instance is in the `Detached` state when it was previously in a `Managed` state but is no longer associated with an `EntityManager`/`Session`.
- **Characteristics:**
  - It still represents a row in the database, and it has a persistent identity.
  - Changes made to a detached entity's fields are *not* automatically synchronized with the database.
  - It retains its data, but is outside the current persistence context.
- **Transition:**

    - From `Managed`: When `EntityManager`/`Session` is closed, `clear()`, or `evict(entity)` is called, or after a transaction commits (if the entity is not returned by the `EntityManager`).
    - To `Managed` state: By calling `entityManager.merge(entity)` or `session.update(entity)/saveOrUpdate(entity)`.
  - **Example:**

```
Product p = entityManager.find(Product.class, 1L); // p is Managed
entityManager.detach(p); // p is now Detached
// Or after a transaction commits, if 'p' is not retained by the EM.
```

4. **Removed State:**

  - **Description:** An entity instance is in the `Removed` state when it is associated with an `EntityManager`/`Session` and has been marked for deletion from the database.
  - **Characteristics:**
    - It is logically managed, but designated for removal.
    - It still has a persistent identity.
    - No longer represents a valid row in the database after the transaction commits.
  - **Transition:**
    - From `Managed`: By calling `entityManager.remove(entity)`.
    - To `New` (Transient): If `remove()` is called on a `New` entity (which is usually an error or no-op).
    - To `Managed`: If `entityManager.persist(entity)` is called on a `Removed` entity before transaction commit, it can revert to `Managed`.
  - **Example:**

```
Product pToDelete = entityManager.find(Product.class, 1L); // pToDelete
is Managed
entityManager.remove(pToDelete); // pToDelete is now in Removed state
// On transaction commit, the corresponding row will be deleted from
the database.
```

Visual Representation:

```
[New (Transient)]
    |
    |  persist() / save()
    V
[Managed (Persistent)] <---- merge() / update() ---- [Detached]
    |                                                  ^
    | remove()                                         |  close() / clear() /
commit (no longer referenced)
    V                                                  |
[Removed] ------------------------------------------
    |
    | commit() (DB delete)
```

```
    V
(Garbage Collected)
```

Understanding these states helps in predicting how changes to entity objects will be synchronized with the database and managing their lifecycle efficiently.

16. How have you implemented many-to-many relations in your project? Explain code.

(This answer needs to be adapted to your actual project if you have one. I'll provide a common implementation using JPA with a join table.)

In my project, I implemented many-to-many relationships using a **join table** in the database, and represented this in JPA using the `@ManyToMany` annotation. This is the standard and most flexible approach for many-to-many relationships in a relational database context with ORM frameworks like Hibernate/JPA.

Scenario: Let's consider a project management system where `Projects` can have multiple `Developers`, and a `Developer` can work on multiple `Projects`.

Database Schema (Conceptual):

```sql
-- projects table
CREATE TABLE projects (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    -- other project details
);

-- developers table
CREATE TABLE developers (
    id BIGINT PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL
    -- other developer details
);

-- project_developer (JOIN TABLE)
CREATE TABLE project_developer (
    project_id BIGINT NOT NULL,
    developer_id BIGINT NOT NULL,
    PRIMARY KEY (project_id, developer_id), -- Composite Primary Key
    FOREIGN KEY (project_id) REFERENCES projects(id) ON DELETE CASCADE,
    FOREIGN KEY (developer_id) REFERENCES developers(id) ON DELETE CASCADE
);
```

JPA Entity Implementation:

For a many-to-many relationship, you typically need to decide which side "owns" the relationship (i.e., which side manages the join table). The owning side defines the `@JoinTable` annotation, while the inverse side uses `mappedBy`.

1. `Project.java` **(Owning Side):**

```
package com.example.projectmgmt.model;

import jakarta.persistence.*;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;

import java.util.HashSet;
import java.util.Set;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Project {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    // Many-to-Many relationship with Developer
    // This is the owning side because it defines @JoinTable
    @ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE}) // Cascade
operations
    @JoinTable(
        name = "project_developer", // Name of the join table
        joinColumns = @JoinColumn(name = "project_id"), // Foreign key for Project
in join table
        inverseJoinColumns = @JoinColumn(name = "developer_id") // Foreign key for
Developer in join table
    )
    private Set<Developer> developers = new HashSet<>(); // Use Set to ensure
uniqueness

    // Helper methods to manage the relationship on both sides
    public void addDeveloper(Developer developer) {
        this.developers.add(developer);
        developer.getProjects().add(this); // Ensure inverse side is also updated
    }

    public void removeDeveloper(Developer developer) {
        this.developers.remove(developer);
        developer.getProjects().remove(this); // Ensure inverse side is also
updated
    }

    // Constructor without ID for new projects
    public Project(String name) {
        this.name = name;
```

```
    }
}
```

2. `Developer.java` **(Inverse Side):**

```
package com.example.projectmgmt.model;

import jakarta.persistence.*;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;

import java.util.HashSet;
import java.util.Set;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
public class Developer {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @Column(unique = true)
    private String email;

    // Many-to-Many relationship with Project
    // This is the inverse side because it uses mappedBy
    @ManyToMany(mappedBy = "developers") // "developers" is the field name in
Project entity
    private Set<Project> projects = new HashSet<>(); // Use Set to ensure
uniqueness

    // Helper methods to manage the relationship on both sides
    public void addProject(Project project) {
        this.projects.add(project);
        project.getDevelopers().add(this); // Ensure owning side is also updated
    }

    public void removeProject(Project project) {
        this.projects.remove(project);
        project.getDevelopers().remove(this); // Ensure owning side is also
updated
    }

    // Constructor without ID for new developers
    public Developer(String name, String email) {
        this.name = name;
```

```
        this.email = email;
    }
}
```

Key Points in the Code:

- `@ManyToMany`**:** Declares the many-to-many relationship on both sides.
- `@JoinTable` **(Owning Side):**
  - `name = "project_developer"`: Specifies the name of the intermediary join table in the database.
  - `joinColumns = @JoinColumn(name = "project_id")`: Defines the foreign key column in the `project_developer` table that refers to the `Project` entity's primary key.
  - `inverseJoinColumns = @JoinColumn(name = "developer_id")`: Defines the foreign key column in the `project_developer` table that refers to the `Developer` entity's primary key.
- `mappedBy = "developers"` **(Inverse Side):**
  - This attribute tells JPA that the `projects` field in the `Developer` entity is the inverse side of the relationship. The `developers` attribute in the `Project` entity is the owning side.
  - The value "developers" must exactly match the name of the `Set<Developer>` field in the `Project` class.
  - The inverse side does not manage the join table; it simply uses the mapping defined by the owning side.
- `HashSet`**:** Using `java.util.Set` (specifically `HashSet`) is crucial for many-to-many relationships to prevent duplicate entries in the join table.
- **Helper Methods (**`addDeveloper`**/**`removeDeveloper` **or** `addProject`**/**`removeProject`**):** It's highly recommended to create helper methods on *both sides* of the relationship to ensure that when you add/remove an entity from one side's collection, the inverse side's collection is also updated. This maintains data consistency in memory and prevents unexpected behavior, especially when managing relationships in business logic *before* persistence.
- `CascadeType`**:** `CascadeType.PERSIST` and `CascadeType.MERGE` are used on the owning side. This means that if you persist a `Project`, any new `Developer`s added to its `developers` set will also be persisted. If you merge a detached `Project`, associated new or detached `Developer`s will also be merged. For `DELETE`, `CascadeType.REMOVE` is an option, but often it's handled manually or via database `ON DELETE CASCADE` if the developers are shared across projects.

Usage Example in a Service/Repository:

```
// Example usage in a Spring Data JPA service
@Service
public class ProjectManagementService {

    @Autowired
    private ProjectRepository projectRepository;

    @Autowired
    private DeveloperRepository developerRepository;

    public Project createProjectWithDevelopers(String projectName, List<Long>
developerIds) {
```

```
        Project project = new Project(projectName);
        // Find existing developers or create new ones
        for (Long devId : developerIds) {
            Developer developer = developerRepository.findById(devId)
                .orElseThrow(() -> new RuntimeException("Developer not found: " +
devId));
            project.addDeveloper(developer); // Use helper method
        }
        return projectRepository.save(project); // Save the project (and related
developers if cascaded)
    }

    public void removeDeveloperFromProject(Long projectId, Long developerId) {
        Project project = projectRepository.findById(projectId)
            .orElseThrow(() -> new RuntimeException("Project not found: " +
projectId));
        Developer developer = developerRepository.findById(developerId)
            .orElseThrow(() -> new RuntimeException("Developer not found: " +
developerId));

        project.removeDeveloper(developer); // Use helper method
        projectRepository.save(project); // Save the changes
    }
}
```

This structured approach with helper methods ensures that the in-memory state of your entities always accurately reflects the relationship, making persistence consistent and reliable.

17. How can we call stored procedures in JDBC, hibernate, and Spring Data JPA?

Calling stored procedures is possible at different layers of a Java application, each offering varying levels of abstraction.

1. Calling Stored Procedures in **JDBC**

JDBC provides the lowest level of abstraction, giving you full control. You use `CallableStatement`.

Example: Suppose you have a stored procedure `GET_EMPLOYEE_DETAILS(IN emp_id INT, OUT emp_name VARCHAR, OUT emp_salary DECIMAL)`

```
import java.sql.*;

public class StoredProcedureJDBC {

    private static final String DB_URL = "jdbc:mysql://localhost:3306/mydb";
    private static final String USER = "user";
    private static final String PASS = "password";

    public static void main(String[] args) {
        try (Connection conn = DriverManager.getConnection(DB_URL, USER, PASS)) {
            // Create a dummy procedure for demonstration (replace with your
```

```
actual DB setup)
            try (Statement stmt = conn.createStatement()) {
                stmt.execute("DROP PROCEDURE IF EXISTS GET_EMPLOYEE_DETAILS");
                stmt.execute("CREATE PROCEDURE GET_EMPLOYEE_DETAILS(IN emp_id INT,
OUT emp_name VARCHAR(255), OUT emp_salary DECIMAL(10,2)) " +
                             "BEGIN " +
                             "    SELECT name, salary INTO emp_name, emp_salary
FROM employees WHERE id = emp_id; " +
                             "END");
                stmt.execute("CREATE TABLE IF NOT EXISTS employees (id INT PRIMARY
KEY, name VARCHAR(255), salary DECIMAL(10,2));");
                stmt.execute("INSERT INTO employees (id, name, salary) VALUES (1,
'Alice', 60000.00);");
                stmt.execute("INSERT INTO employees (id, name, salary) VALUES (2,
'Bob', 75000.00);");
            }

            // Prepare the CallableStatement
            String sql = "{CALL GET_EMPLOYEE_DETAILS(?, ?, ?)}"; // Placeholders
for IN, OUT, OUT parameters
            try (CallableStatement callableStatement = conn.prepareCall(sql)) {

                // Set IN parameter
                callableStatement.setInt(1, 1); // emp_id = 1

                // Register OUT parameters
                callableStatement.registerOutParameter(2, Types.VARCHAR); //
emp_name
                callableStatement.registerOutParameter(3, Types.DECIMAL); //
emp_salary

                // Execute the stored procedure
                callableStatement.execute();

                // Retrieve OUT parameter values
                String empName = callableStatement.getString(2);
                double empSalary = callableStatement.getDouble(3);

                System.out.println("Employee Details:");
                System.out.println("ID: 1, Name: " + empName + ", Salary: " +
empSalary);

            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

2. Calling Stored Procedures in **Hibernate (JPA)**

JPA (and by extension Hibernate) provides `@NamedStoredProcedureQuery` for mapping stored procedures to entities or for calling them directly.

Example: (Same stored procedure as above)

a) Using `@NamedStoredProcedureQuery` (JPA 2.1+):

Define the stored procedure mapping on an entity or in `orm.xml`.

```
import jakarta.persistence.*;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;

@Entity
@Table(name = "employees")
@Data
@NoArgsConstructor
@AllArgsConstructor
@NamedStoredProcedureQuery(
    name = "getEmployeeDetails", // A name to refer to this procedure
    procedureName = "GET_EMPLOYEE_DETAILS", // Actual name of the DB procedure
    parameters = {
        @StoredProcedureParameter(mode = ParameterMode.IN, name = "emp_id", type =
Integer.class),
        @StoredProcedureParameter(mode = ParameterMode.OUT, name = "emp_name",
type = String.class),
        @StoredProcedureParameter(mode = ParameterMode.OUT, name = "emp_salary",
type = Double.class)
    }
)
public class Employee {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;
    private String name;
    private Double salary;
}
```

Then call it via `EntityManager`:

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.ParameterMode;
import jakarta.persistence.Persistence;
import jakarta.persistence.StoredProcedureQuery;

public class StoredProcedureJPA {

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("my-
```

```
persistence-unit"); // Configure persistence.xml
        EntityManager em = emf.createEntityManager();

        em.getTransaction().begin();
        // Insert dummy data
        em.persist(new Employee(1, "Alice", 60000.00));
        em.persist(new Employee(2, "Bob", 75000.00));
        em.getTransaction().commit();


        em.getTransaction().begin();
        StoredProcedureQuery query =
em.createNamedStoredProcedureQuery("getEmployeeDetails");

        // Set IN parameter
        query.setParameter("emp_id", 1);

        // Execute
        query.execute();

        // Get OUT parameters
        String empName = (String) query.getOutputParameterValue("emp_name");
        Double empSalary = (Double) query.getOutputParameterValue("emp_salary");

        System.out.println("Employee Details (JPA):");
        System.out.println("ID: 1, Name: " + empName + ", Salary: " + empSalary);

        em.close();
        emf.close();
    }
}
```

Note: This requires a `persistence.xml` setup for the JPA unit.

b) Using Native SQL Query (Hibernate specific via `Session` or JPA via `EntityManager.createNativeQuery`):

This is more like JDBC, directly executing the call.

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.EntityManagerFactory;
import jakarta.persistence.Persistence;
import org.hibernate.Session;
import org.hibernate.query.NativeQuery; // Hibernate specific query interface

public class StoredProcedureHibernateNative {

    public static void main(String[] args) {
        EntityManagerFactory emf = Persistence.createEntityManagerFactory("my-
persistence-unit");
        EntityManager em = emf.createEntityManager();
        Session session = em.unwrap(Session.class); // Get Hibernate Session from
```

```
JPA EntityManager

        // Insert dummy data (same as above)
        em.getTransaction().begin();
        em.persist(new Employee(1, "Alice", 60000.00));
        em.persist(new Employee(2, "Bob", 75000.00));
        em.getTransaction().commit();

        session.beginTransaction();
        NativeQuery<?> query = session.createNativeQuery("{CALL
GET_EMPLOYEE_DETAILS(:empId, :empName, :empSalary)}");

        query.setParameter("empId", 1);
        query.addSynchronizedEntityClass(Employee.class); // Optional: to
synchronize entity state

        // For OUT parameters, you might need to use JDBC connection directly or
handle result sets.
        // Hibernate's NativeQuery for OUT parameters can be tricky, often
requiring ResultSet handling.
        // Alternatively, use parameter binding for stored functions or procedures
returning result sets.

        // For simple OUT params, often execute and then retrieve results from the
CallableStatement unwrapped
        // from the Hibernate connection, or map procedure output as a result set
if the procedure returns one.
        // A more common pattern for OUT parameters with NativeQuery is if the
procedure returns a result set.

        // Simpler for procedures that return a result set:
        // NativeQuery<Employee> query = session.createNativeQuery("CALL
GET_ALL_EMPLOYEES()", Employee.class);
        // List<Employee> employees = query.getResultList();

        // For IN/OUT/OUT, it's often more straightforward to use
EntityManager.createStoredProcedureQuery
        // directly as shown in the JPA section above, or fall back to plain JDBC.
        session.getTransaction().commit();

        // For procedures returning results directly (common with JPA criteria):
        StoredProcedureQuery spQuery =
em.createStoredProcedureQuery("GET_EMPLOYEE_DETAILS");
        spQuery.registerStoredProcedureParameter("emp_id", Integer.class,
ParameterMode.IN);
        spQuery.registerStoredProcedureParameter("emp_name", String.class,
ParameterMode.OUT);
        spQuery.registerStoredProcedureParameter("emp_salary", Double.class,
ParameterMode.OUT);

        spQuery.setParameter("emp_id", 1);
        spQuery.execute();
        System.out.println("Employee Details (JPA StoredProcedureQuery
```

```
directly):");
        System.out.
```