1) What is IOC ?(Inversion of Control)?
IOC means:
You don’t create objects, Spring creates and manages them for you.

Simple words:
Normally you control object creation

In Spring, control is given to the Spring container

Example:
Instead of:
Car car = new Car();
Spring does it for you.
 IOC = Spring controls object lifecycle

IOC means the control of creating and managing objects is given to
the Spring container instead of the programmer. In normal Java programs,
 we create objects using the new keyword, but in Spring, the framework creates,
manages, and destroys
 the objects for us. This makes the application loosely coupled and easy to
maintain.

2) what is dependency injection ?
Dependency Injection is a way to implement IOC.
Simple words:
spring injects required objects (dependencies) into a class automatically.

Example:
```
class Car {
   Engine engine;  // dependency
}
```

Spring provides Engine to Car.
DI = Giving dependent objects automatically

Dependency Injection is a design pattern used to implement IOC. It means Spring
automatically provides
the required dependent objects to a class instead of the class creating them itself.
By injecting dependencies through constructor,
setter, or field injection, the code becomes more flexible, reusable, and easy to
test.

3)what is autowiring in spring ?
Autowiring means Spring automatically injects dependencies without manual
configuration.
Spring finds and injects the required object by itself.

Example:
```
@Autowired
Engine engine;
```

Types of autowiring:
By type (most common)
By name
Constructor injection (recommended)

Autowiring = Automatic dependency injection

Autowiring is a feature in Spring where the framework automatically injects the required dependency
into a class without explicit configuration. Spring identifies the dependency by type or name and injects it using annotations
like @Autowired. This reduces boilerplate code and simplifies dependency management.

4)@Qualifer vs @primary annations
Used when multiple beans of same type exist.

Problem:
@Autowired
Payment payment;   // Which

@Primary

 Default bean
example

@Primary
@Component
class UpiPayment implements Payment {}

Spring chooses this by default.

@Qualifier

Explicitly tell Spring which bean to use

@Autowired
@Qualifier("cardPayment")
Payment payment;

@Qualifier overrides @Primary

When multiple beans of the same type are present, Spring gets
confused about which one to inject. @Primary is used to mark one bean
as the default choice, so Spring will use it automatically. @Qualifier
is used when we want to specifically tell Spring which bean to inject by name. If both are used, @Qualifier has higher priority and overrides @Primary.

5)What is @Component , @Bean and @Configuration

@Component

Used on class

```
@Component
class ServiceA {}
```

Spring auto-detects it
Used for your own classes

@Bean

👉 Used on method

```
@Bean
public ServiceA serviceA() {
   return new ServiceA();
}
```

Used when you cannot modify the class
Often used for third-party libraries

@Configuration

👉 Used on class

```
@Configuration
class AppConfig {}
```

Holds @Bean methods
Tells Spring this is a configuration class

@Component is used on a class to tell Spring to automatically
create and manage its object as a bean. @Bean is used on a method
to manually create a bean, usually when we are using third-party classes
that we cannot modify. @Configuration is used on a class to indicate that it
contains
 @Bean methods and is responsible for defining Spring beans.

6)@Controller vs @Service vs @Repository

```
@Controller
public class UserController { }

@Service
public class UserService { }

@Repository
public class UserRepository { }
```

👉 Extra point:
@Repository also converts database exceptions into Spring’s DataAccessException.

@Controller is used in the presentation layer to handle incoming HTTP requests and return views or responses.
 @Service is used in the business logic layer where actual processing and rules are written.
@Repository is used in the data access layer to interact with the database. It also helps in translating database-related exceptions into Spring exceptions.
All three are special types of @Component and help in separating application layers clearly.

7)@Controller vs @RestController

```
@Controller
public class PageController {
    @ResponseBody
    @GetMapping("/msg")
    public String hello() {
        return "Hello";
    }
}

@RestController
public class ApiController {
    @GetMapping("/msg")
    public String hello() {
        return "Hello";
    }
}
```

👉 Interview line:
@RestController = @Controller + @ResponseBody

@Controller is mainly used in traditional Spring MVC applications and usually returns a view (like JSP or Thymeleaf).
 @RestController is used in REST APIs and directly returns data such as JSON or XML.

@RestController internally combines @Controller and @ResponseBody, so we don’t need to add @ResponseBody to every method.

8)@PostMapping , @GetMapping, @PutMapping ,@PatchMapping and @DeleteMapping
These annotations are used to map HTTP requests to controller methods.
 @GetMapping is used to fetch data from the server. @PostMapping is used to create new data.
@PutMapping is used to update an entire existing resource. @PatchMapping is used to update only specific fields of a resource.
@DeleteMapping is used to delete data from the server

9)@Putmapping vs @PatchMapping

```
// PUT
{
  "id": 1,
```

```
  "name": "Nisarga",
  "email": "nisarga@gmail.com"
}
java
Copy code
// PATCH
{
  "email": "newmail@gmail.com"
}
```
☞ Interview tip:
Use PUT for full update, PATCH for partial update.

@PutMapping is used when we want to update the complete object, meaning all fields are replaced with new values.
@PatchMapping is used for partial updates, where only selected fields are modified and the remaining fields stay unchanged.
In simple terms, PUT is a full update, while PATCH is a partial update.

10)how to handle the excepions in Spring or Springboot
(@ControllerAdvice,@ExceptionHandler
A) @ExceptionHandler (Controller-level)

Handles exceptions inside a single controller.

```
@ExceptionHandler(NullPointerException.class)
public String handleException() {
    return "error";
}
```

B) @ControllerAdvice (Global exception handling)

Used to handle exceptions across all controllers.

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(Exception.class)
    public ResponseEntity<String> handleAll(Exception ex) {
        return new ResponseEntity<>(ex.getMessage(),
HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

C) @RestControllerAdvice

Same as @ControllerAdvice but returns JSON response.

```
@RestControllerAdvice
public class GlobalExceptionHandler {
```

```
    @ExceptionHandler(RuntimeException.class)
    public ResponseEntity<String> handle(RuntimeException ex) {
        return new ResponseEntity<>(ex.getMessage(), HttpStatus.BAD_REQUEST);
    }
}
```

In Spring or Spring Boot, exceptions are handled globally using @ControllerAdvice and @ExceptionHandler.
@ControllerAdvice is used to create a global exception handling class that applies to all controllers.
@ExceptionHandler is used inside this class to handle specific exceptions and return a proper response.

11) What are the bean types in Spring
In Spring, beans are objects managed by the Spring IoC container.
Based on how they are created and managed,
Spring beans are mainly of two types:
Singleton (default) – Only one instance of the bean is created for the entire Spring container.
Every request for that bean returns the same object.
This is most commonly used for service and repository classes.
Prototype – A new instance of the bean is created every time it is requested from the container.
This is useful when the bean is stateful.

Spring also provides other specialized bean scopes like request, session, application, and websocket, which are mainly used in web applications.

example

```
@Component
@Scope("singleton")
public class UserService { }
```

Singleton is default, no need to specify it.

12)Explain the bean life cycle in Spring
The Spring bean life cycle describes what happens to a bean from the time it is created until it is destroyed.
First, the Spring container creates the bean instance. Then it injects dependencies using constructor or setter injection.
After that, Spring calls Aware interfaces (like BeanNameAware) if implemented.
Next, BeanPostProcessor methods are executed before initialization.
Then the bean's initialization methods are called (@PostConstruct or afterPropertiesSet()). After this, the bean is ready to use. When the application shuts down,
Spring calls destroy methods (@PreDestroy or destroy()), allowing cleanup of resources.

example

Create → Inject → Initialize → Use → Destroy

```
@Component
public class DemoBean {

    @PostConstruct
    public void init() {
        System.out.println("Bean initialized");
    }

    @PreDestroy
    public void destroy() {
        System.out.println("Bean destroyed");
    }
}
```

13)SetterInjection vs Constructor injection

Setter Injection

Dependency injected using setter method

Dependency can be optional

Not recommended for mandatory fields

```
@Component
public class Car {

    private Engine engine;

    @Autowired
    public void setEngine(Engine engine) {
        this.engine = engine;
    }
}
```

Constructor Injection

Dependency injected using constructor

Dependency is mandatory

Best practice 👍

```
@Component
public class Car {

    private final Engine engine;
```

```
    @Autowired
    public Car(Engine engine) {
        this.engine = engine;
    }
}
```

Constructor injection happens when dependencies are provided through the constructor, while setter injection
provides dependencies using setter methods. Constructor injection is preferred because it ensures that all required
dependencies are available at object creation time, making the bean immutable and safe. Setter injection is useful when
dependencies are optional or need to be changed later. Constructor injection avoids NullPointerException and is better for testing,
while setter injection gives more flexibility but may lead to partially initialized objects.

14)form validations in spring
Spring uses JSR-380 Bean Validation (Hibernate Validator).
Steps for Form Validation

Add validation dependency
(Spring Boot includes it by default)
Use validation annotations in Model

Model Example:

```
public class User {

    @NotBlank(message = "Name is required")
    private String name;

    @Email(message = "Invalid email")
    private String email;
}
```

Controller Example:

```
@PostMapping("/register")
public String register(
        @Valid @ModelAttribute User user,
        BindingResult result) {

    if (result.hasErrors()) {
        return "register";
    }
    return "success";
}
```

Form validation in Spring ensures that user input is correct before processing it.
Spring uses Bean Validation (JSR-380) with annotations like @NotNull, @NotEmpty,

@Size, @Email, and @Pattern.
These annotations are placed on model class fields. In the controller, the @Valid annotation is used to trigger validation,
and BindingResult is used to capture validation errors. If errors exist, the form is shown again with error messages; otherwise,
 the data is processed. This approach helps prevent invalid data from reaching the database.

15)RequestParam vs Pathparam annotations
@RequestParam

Reads value from query parameter

Optional or required

Used for filtering/searching

URL:
/users?id=10

```
@GetMapping("/users")
public String getUser(@RequestParam int id) {
    return "User ID: " + id;
}
```

@PathVariable

Reads value from URL path

Mostly mandatory

Used in REST APIs

URL:
/users/10

```
@GetMapping("/users/{id}")
public String getUser(@PathVariable int id) {
    return "User ID: " + id;
}
```

@RequestParam is used to extract query parameters from the URL, usually optional or used for filtering and searching. For example:
/users?age=25 → age is a request parameter.

@PathVariable is used to extract values directly from the URL path, usually mandatory and used to identify a specific resource. For example:
/users/10 → 10 is a path variable.

In short, @RequestParam is commonly used for optional data, while @PathVariable is

used for RESTful URLs that represent resources.

16) How to configure the sever in spring
In Spring Boot
Spring Boot uses an embedded server (Tomcat by default), so no external server configuration is required.
You configure the server in application.properties or application.yml.

Example:

```
server.port=8081
server.servlet.context-path=/myapp
```

server.port → changes the port number
context-path → sets application base URL
To change server (Tomcat → Jetty):
xml

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

In Spring (without Boot)
Configure external Tomcat

Use web.xml and DispatcherServlet configuration

In Spring (especially Spring Boot), server configuration is very easy because Spring Boot
provides an embedded server like Tomcat by default, so we don’t need to install or configure a server separately.
The server settings are configured using the application.properties or application.yml file. For example, we can
change the server port using server.port=8081 or set a context path using server.servlet.context-path=/app. Spring Boot automatically
starts the server when the application runs. In traditional Spring (without Boot), we need to configure an external server like Apache Tomcat,
create a WAR file, define configurations in web.xml, and deploy the application manually on the server. Spring Boot simplifies this entire process.

17) Explain Spring MVC flow
Spring MVC follows the Front Controller design pattern.

Flow:
Client sends HTTP request
Request goes to DispatcherServlet
DispatcherServlet finds the Controller
Controller calls Service layer
Service interacts with DAO / Database
Controller returns Model & View

View Resolver selects the view
Response sent back to client
DispatcherServlet is the heart of Spring MVC.

Spring MVC works based on the Model–View–Controller (MVC) design pattern. When a client sends a request,
it first goes to the DispatcherServlet, which acts as the front controller. The DispatcherServlet checks
the HandlerMapping to find which controller method should handle the request. The controller processes the request,
interacts with the service layer, and returns a Model (data) and a View name. The DispatcherServlet then uses a ViewResolver
to locate the actual view (like JSP or Thymeleaf). Finally, the view renders the data and sends the response back to the client.
This flow ensures a clear separation between business logic, presentation logic, and request handling.

18) What is Spring ORM
Spring ORM is a module that integrates Spring with ORM frameworks like:
Hibernate
JPA
MyBatis
Purpose:
Simplifies database access
Handles transactions
Converts database exceptions into Spring exceptions
Benefits:
Less boilerplate code
Easy transaction management
Loose coupling with ORM tools
Spring ORM does not replace Hibernate, it supports it.

Spring ORM (Object Relational Mapping) is a module in Spring that helps integrate Spring applications with ORM frameworks like Hibernate,
JPA, and MyBatis. It simplifies database operations by reducing boilerplate code and handling resource management, transaction management,
and exception handling automatically. Spring ORM allows developers to work with Java objects instead of writing complex SQL queries. It also
provides consistent exception handling by converting database-specific exceptions into Spring's unified DataAccessException. This makes database
interaction easier, cleaner, and more maintainable.

19)prototype vs Singleton bean and how it works
Singleton Bean (Default)
Only one object per Spring container
Same instance shared everywhere

@Component
@Scope("singleton")
Use case: Service, Repository classes

Prototype Bean
New object created every time it is requested

@Component
@Scope("prototype")
Use case: Objects with state (temporary data)

In Spring, Singleton is the default bean scope, where only one object of a bean is created per Spring container,
and the same instance is shared across the application. Whenever the bean is requested, Spring returns the same object. In contrast,
Prototype scope creates a new object every time the bean is requested. Singleton beans are created at application startup, while prototype beans
 are created only when requested. Singleton is best suited for stateless beans like services, whereas prototype is useful when each request needs a separate object.
 Spring manages the full lifecycle of singleton beans, but for prototype beans, Spring only creates the object and does not manage its destruction.

20)What is Spring Security
Spring Security is a separate Spring framework module used for securing applications.
It provides:
Authentication (who are you?)
Authorization (what can you access?)
Login/Logout
Role-based access
JWT, OAuth2 support
CSRF protection

Example:
@EnableWebSecurity
class SecurityConfig {}

Spring Security is a powerful framework used to secure Spring applications by providing authentication and authorization features.
Authentication verifies who the user is, while authorization checks what the user is allowed to access. Spring Security
basic authentication, JWT, OAuth, and LDAP. It also helps protect applications against common security threats like CSRF,
session fixation, and clickjacking. By using Spring Security, developers can easily secure REST APIs and web applications with minimal configuration.

spring vs springboot
Spring is a framework that gives full control but needs manual configuration and an external server like Tomcat.
Spring Boot is built on top of Spring and makes development easy and fast by providing auto-configuration, embedded servers, and less setup.