**1. \*\*Java Fundamentals\*\*:  
   - Explain the concept of Java Generics. Why are they used, and how do they improve code quality?**

Generics in Java provide a way to create classes, interfaces, and methods that can operate on different types without sacrificing type safety. The main idea is to allow the use of a placeholder for a data type, which is specified when the code is instantiated or called. This enables the creation of more flexible and reusable code.

Why are they used?

Type Safety:

Generics provide compile-time type checking, ensuring that the code is type-safe. This means that the compiler can detect and prevent type-related errors before the code is run, reducing the chances of runtime errors.

Code Reusability:

Generics allow you to write code that can work with a variety of data types. This promotes code reusability since the same code structure can be used with different types of data without duplication.

**Elimination of Type Casting:**

Prior to generics, when working with collections (e.g., **List**, **Map**), elements were stored as **Object** types, leading to the need for explicit type casting when retrieving elements. Generics eliminate the need for casting by providing type information at compile time.

Readability and Maintainability:

Generics make code more readable by explicitly specifying the types used in a class, method, or interface. This makes the code self-documenting and easier to understand. It also reduces the chance of errors related to incorrect type usage.

Algorithm Abstraction:

Generics allow you to write algorithms and data structures that are independent of specific data types. For example, you can write a generic sorting algorithm that works for any comparable type without rewriting the algorithm for each data type.

How do they improve code quality?

Compile-Time Checks:

Generics provide compile-time type checks, allowing the compiler to catch type-related errors early in the development process. This results in more robust code and reduces the likelihood of bugs.

Code Reusability and Reduction of Redundancy:

Generics promote the reuse of code across different data types, reducing redundancy in the codebase. This makes the codebase more maintainable and easier to update.

Readability and Expressiveness:

Generics make code more expressive and readable by clearly specifying the types involved. This helps developers understand the intended usage of the code and reduces the need for comments to explain the type relationships.

Prevention of Type-Related Runtime Errors:

With generics, many type-related errors are caught at compile time, preventing them from causing runtime issues. This contributes to the overall stability and reliability of the software.

**Discuss the differences between checked and unchecked exceptions in Java.**

Checked Exceptions:

Compile-Time Checking:

Checked exceptions are checked at compile time. This means that the Java compiler ensures that the code either handles these exceptions using a try-catch block or declares that it throws the exception using the throws clause in the method signature.

Subclass of Exception:

Checked exceptions are subclasses of the Exception class (excluding subclasses of RuntimeException and its subclasses). Examples include IOException, SQLException, and FileNotFoundException.

Forces Exception Handling:

Code that may throw a checked exception must explicitly handle it or declare it in the method signature using the throws clause. This forces the developer to handle potential exceptions, making the code more robust.

Intended for Recoverable Situations:

Checked exceptions are typically used for situations where recovery is possible, such as file I/O operations, database operations, and network operations. They indicate conditions that are beyond the program's control.

Unchecked Exceptions:

Runtime Checking:

Unchecked exceptions are not checked at compile time. They occur at runtime, and the compiler does not enforce the use of try-catch blocks or the declaration of throws for unchecked exceptions.

Subclass of RuntimeException:

Unchecked exceptions are subclasses of RuntimeException or its subclasses. Examples include NullPointerException, ArrayIndexOutOfBoundsException, and ArithmeticException.

Optional Exception Handling:

Developers are not forced to handle or declare unchecked exceptions. While it is good practice to handle them, it's not a requirement. Unchecked exceptions are often used to indicate programming errors, such as dividing by zero or accessing an array index out of bounds.

Intended for Unrecoverable Situations:

Unchecked exceptions are typically used for situations where recovery may not be possible or practical. They often indicate errors in the program logic that should be fixed during development.

2. \*\*Spring Boot Fundamentals\*\*:  
   - Describe the Spring Boot auto-configuration mechanism.  
   - Explain the role of the `@RestController` annotation in a Spring Boot application.

**Spring Boot Auto-Configuration Mechanism:**

Spring Boot's auto-configuration mechanism is a key feature that simplifies the configuration of Spring applications. It allows developers to automatically configure the application based on the dependencies in the classpath. This feature aims to reduce the need for boilerplate configuration code and enable developers to get started quickly with minimal manual setup.

Here's how the Spring Boot auto-configuration mechanism works:

**Classpath Scanning:**

Spring Boot scans the classpath for libraries and frameworks commonly used in various types of applications, such as databases, web servers, messaging systems, etc.

**Conditional Configuration:**

Auto-configuration classes in Spring Boot use conditions to determine whether a particular configuration should be applied. Conditions are based on the presence or absence of certain classes, beans, properties, etc. If the conditions are met, the auto-configuration is activated.

**Property-Based Configuration:**

Auto-configuration often relies on properties defined in the **application.properties** or **application.yml** files. These properties provide a way to customize the auto-configured beans and components.

**Customization and Overrides:**

Developers can customize or override the auto-configuration by providing their own configuration. This can be achieved through additional configuration classes or properties. Spring Boot's auto-configuration is designed to be flexible and easily adaptable to specific project requirements.

**Example:**

As an example, if Spring Boot detects the presence of the H2 database library in the classpath, it will automatically configure a datasource bean for H2. This eliminates the need for explicit datasource configuration in most cases.

**Role of @RestController Annotation in a Spring Boot Application:**

The **@RestController** annotation is a specialized version of the **@Controller** annotation in Spring. It is specifically designed for building RESTful web services. When applied to a class, the **@RestController** annotation indicates that the class is a controller responsible for handling HTTP requests and returning appropriate responses, typically in JSON or XML format.

Key features of **@RestController**:

**Combination of @Controller and @ResponseBody:**

The **@RestController** annotation is a combination of **@Controller** and **@ResponseBody**. While a standard **@Controller** returns a view, a **@RestController** returns the object and the object data is directly written into the HTTP response as JSON or XML.

**Simplifies RESTful Endpoint Development:**

Using **@RestController** eliminates the need to annotate every method with **@ResponseBody**. This makes the development of RESTful endpoints more concise and easier to read.

**Example:**

In a Spring Boot application, a simple RESTful controller might look like this:

In this example, the **@RestController** annotation is applied to the class, and the methods annotated with **@GetMapping** handle HTTP GET requests. The response is automatically converted to JSON or XML, based on the content negotiation settings.

In summary, **@RestController** plays a crucial role in Spring Boot applications by simplifying the development of RESTful endpoints. It combines the **@Controller** and **@ResponseBody** annotations, making it easy to create controllers that produce JSON or XML responses for RESTful services.