भारतीय प्रौद्योगिकी संस्थान मुंबई
पवई, मुंबई - 400 076, भारत
Indian Institute of Technology Bombay
Powai, Mumbai - 400 076, India
(उच्चतर शिक्षा शिभाग, शिक्षा मंत्रालय, भारत सरकार के तहत एक स्वायत्त संगठन)
(An Autonomous Organization under the Department of Higher Education, Ministry of Education, Government of India)

Recruitment Unit
Administration Section
Phone : (+91-22) 2159 6171
(+91-22) 2159 6172
E-mail Id - jobs@iitb.ac.in
Website : www.iitb.ac.in

No. Admin/Rect/2025 **Date: 20.02.2025**

Subject: Invitation for feedback/ representation(s), if any, on the Question Paper and the Answer Keys of the written test held on **19.02.2025** for the post of **Technical Officer (Scale-I) (Job Ref. No. 50571745)**.

The Question Paper and the Answer Key of the Written Test conducted for the post of **Technical Officer (Scale-I) (Job Ref. No. 50571745)** on **Wednesday, 19.02.2025,** are enclosed below as **Annexure -1**.

Feedback/ concern(s)/ representation(s), if any, is/are invited from the candidates regarding the answer to any question, on the email id aradmin2@iitb.ac.in latest by **today**, i.e., **on 21.02.2025, before 05:00 p.m.**

By Order

Note: *Candidates must be specific in their representations/ concerns/ feedback and indicate their question(s) and explanation(s), if desired.*

Written Examination for Technical Officer

Date 19 Feb 2025
Duration 3 hrs

Anonymous name

ASC, IIT Bombay

Instructions:

a. Collect a **randomly chosen chit** from the tray. Fill, tear, and deposit one part in the box. Keep the part containing only the Anonymous name with you.

b. **Do not write your name or application ID** anywhere in the question paper or the answer script. Violation of this will lead to disqualification from further evaluation.

c. On the question paper and the answer script, you must **only write the anonymous** name provided to you.

d. Each question may contain **more than one** correct answer.

e. Mark the answers in the **question paper** itself.

f. **Explanation** for the correct and incorrect option must be given **on the answer script** provided. Example:

Q5. Ans: (b), (d)

Explanation:

- (a) is incorrect because...
- (b) is correct because...
- (c) is incorrect because...
- (d) is correct because...

g. Written answer to each question must start in a **fresh page**, with the question number clearly written.

h. **Number** all the pages.

i. **Fill in the cover page** of the answer booklet with the starting page numbers of each question.

Marking pattern:

1. Each question carries 5 marks.
2. Correct option(s) with explanation: 2 marks
3. Incorrect option(s) with explanation: 3 marks
4. Answers without explanation: no marks.
5. There is no negative marking.

---

1. Consider a Spring Boot REST controller that handles requests to create a new resource. Which of the following approaches is the MOST robust and recommended way to handle exceptions that might occur during resource creation (e.g., validation errors, database exceptions, etc.)?

   (a) Using a generic '@ExceptionHandler' method that catches all exceptions and returns a generic error response.

   (b) Catching specific exceptions (e.g., 'MethodArgumentNotValidException', 'DataAccessException') in separate '@ExceptionHandler' methods and returning tailored error responses.

   (c) Using a combination of '@ControllerAdvice' and '@ExceptionHandler' to handle exceptions globally and provide specific error responses based on the exception type.

   (d) Relying solely on Spring Boot's default exception handling mechanism, which automatically converts exceptions to HTTP error responses.

   Solution key

   Correct Answer: (c)

   Explanation:

   Option (c) is the most robust and recommended approach

   - *Centralized Exception Handling:* Keeps exception handling logic in one place.
   - *Specific Error Responses:* Provides more informative and helpful error messages to clients.
   - *Improved Maintainability:* Makes it easier to manage and update exception handling logic.
   - *Better User Experience:* Provides more meaningful feedback to users.

   (a) While a generic '@ExceptionHandler' can catch all exceptions, it's generally not a good practice to return a generic error response for all exceptions. This can hide important details about the error and make it difficult to debug. It's better to provide specific error messages whenever possible.

   (b) Catching specific exceptions in separate '@ExceptionHandler' methods within each controller can lead to code duplication and make it harder to maintain exception handling logic. '@ControllerAdvice' provides a better way to centralize this logic.

   (c) *Correct*

   (d) Relying solely on Spring Boot's default exception handling is not sufficient for production applications. Spring Boot's default handling provides a basic error response, but it often doesn't include enough information for clients. Exception handling needs to be customised to provide more specific error messages and handle different types of exceptions appropriately. It also might expose internal error details, which could be a security risk.

2. Which of the following are likely to be correct about the following fluent interface that calculates the tax on an income:
   `taxCalculator.setIncome(10000).foo(500).multiply(0.25).taxValue()`?

   (a) `setIncome()` returns a TaxCalculator object.

   (b) `foo()` returns `this`

   (c) `taxValue()` returns `this`

   (d) `multiply()` multiplies the previous argument 500 by 0.25.

   Solution key

   Correct answers: (a), (b)

   Explanation:

   - (a) `setIncome()` is likely to return a `TaxCalculator` object to allow method chaining. This is a common pattern in fluent interfaces.

- (b) `foo()` is likely to return `this` to allow method chaining.
- (c) `taxValue()` is likely to return the calculated tax value (a double), not the object TaxCalculator with `this`.
- (d) `multiply()` is likely to apply multiplication to the return object of `foo` using method chaining, and not its argument 500 by 0.25.

3. Consider the following Spring Security configuration:

```
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/public/**").permitAll()
            .requestMatchers("/admin/**").hasRole("ADMIN")
            .anyRequest().authenticated()
        )
        .formLogin(form -> form
            .loginPage("/login")
            .permitAll()
        )
        .and()
        .httpBasic();
    return http.build();
}
```

What is the effect of the `.and()` method call between the `formLogin()` and `httpBasic()` configurations?

Answer Choices:

(a) It separates the form login and HTTP basic authentication configurations, making them mutually exclusive. If form login fails, HTTP basic will be attempted.

(b) It combines the form login and HTTP basic authentication configurations, allowing users to authenticate using either of methods: `formLogin()` OR `httpBasic()`

(c) It has no effect; the `httpBasic()` configuration would be applied regardless of the `.and()` call.

(d) It throws an exception because `.and()` cannot be used between `formLogin()` and `httpBasic()`.

Solution key

Correct Answer: (b)

Explanation:

The `.and()` method in Spring Security's `HttpSecurity` is used to chain different configuration blocks together. Both the blocks are executed, irrespective of whether the first one fails. In this case, it signifies that both form login and HTTP basic authentication are enabled for the application. A user can authenticate using *either* form login (by visiting the `/login` page) *or* HTTP basic (by providing credentials in the Authorization header).

(a) `.and()` does *not* make the authentication methods mutually exclusive. They are both active.

(b) *Correct.*

(c) `.and()` is essential for chaining these configurations. Without it, the `httpBasic()` configuration wouldn't be correctly applied if `formLogin` configuration fails.

(d) `.and()` is perfectly valid and commonly used between authentication methods like `formLogin()` and `httpBasic()`. It doesn't throw an exception in this context.

4. Consider the following Java code snippet:

```
import java.util.function.Function;

public class Example {

    public static void main(String[] args) {
        Function<Integer, String> myFunction = (Integer x) -> {
            return "The number is: " + x;
```

```
        };

        String result = myFunction.apply(10);
        System.out.println(result);
    }
}
```

Which of the following best describes what the following expression does:
`(Integer x)-> { return "The number is: "+ x; }`

Answer Choices:

(a) It takes an integer as input and returns a string representation of that integer.

(b) It defines a pointer to an integer and returns an integer.

(c) It checks if the integer `x` is greater than the value of the expression inside {}.

(d) It defines a function that takes no input and returns a string.

Solution key

Correct Answer: a)

Explanation:

The lambda expression `(Integer x)-> return "The number is: "+ x;` implements the `Function< Integer, String>` interface. This interface specifies a function that takes an 'Integer' as input and returns a 'String'. The lambda expression takes an integer 'x', converts it to a string representation, and concatenates it with the prefix "The number is: ".

(a) *Correct.*

(b) Incorrect. This is a lambda expression not a pointer to an integer.

(c) Incorrect. This is a lambda expression not a comparison operation.

(d) Incorrect. The lambda expression takes an Integer as input.

5. Consider the following Spring Security configuration:

```
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
      http
        .authorizeHttpRequests(authorize -> authorize
            .requestMatchers("/api/**").permitAll()
            .requestMatchers("/public/**").hasAnyRole("ADMIN", "USER")
            .requestMatchers("/api/sensitive/**").hasRole("ADMIN")
            .requestMatchers("/admin/**").hasRole("USER")
            .requestMatchers("/**").authenticated()
        )
        .httpBasic();

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

Which of the following are likely to be true?

(a) `/api/sensitive/info` can be accessed only by "ADMIN"

(b) `/admin/dashboard` can be accessed by "ADMIN"

(c) `/public/data` can be accessed by any role

(d) `/api/dashboard` needs authentication

Solution key

Correct Answer: d)

Explanation:

The order of 'requestMatchers' in Spring Security matters. If a match occurs the execution is skipped to the next block. The last match is a catch-all rule that requires authentication for all endpoints.

(a) Incorrect. since `/api/**` is matched before and has a permitAll, any authenticated user can access `/api/sensitive/info` not just "ADMIN".

(b) Incorrect. `/admin/dashboard` is accessible only to "USER" role.

(c) Incorrect. `/public/**` is accessible only to "ADMIN" or "USER". If there are other roles in the system, they cannot access it.

(d) *Correct.*

6. Consider the following Spring MVC controller:

```
@RestController
@RequestMapping("/api")
public class DataController {

    @GetMapping(value = {"/data", "/info"}, params = {"details=foo"})
    public String getFoo() {
        return "Data Foo";
    }

    @GetMapping(value = {"/data", "/info"})
    public String getBar() {
        return "Bar Data";
    }

    @GetMapping("/data/{details}")
    public String getBaz(@PathVariable Long details) {
        return "Data details: " + details;
    }

    @GetMapping(path = "/api/info", params = {"type=details"})
    public String getBaa() {
        return "Info Baa";
    }

    @GetMapping(path = "/info", params = {"type=details"})
    public String getBat() {
        return "Info Bat";
    }

    @GetMapping("/details")
    public String getDetails() {
        return "Details";
    }
}
```

A request is made to the endpoint `/api/info?type=details`. Which method will handle this request?

Answer Choices:

(a) `getFoo()`

(b) `getBaz(details)`

(c) `getBat()`

(d) `getBaa()`

Solution key

Correct Answer: c)

Explanation: Though there are overlapping paths, the presence of the 'type=details' parameter and the RequestMapping annotation makes the mapping for `getBat()` the correct function to handle the request.

(a) Incorrect. `getFoo` is called only if the details parameter is present as a key, not as a value.

(b) Incorrect. `getBaz` is called only if the path variable `details` is present in the request.

(c) *Correct.*

(d) Incorrect. `getBaa` is called only if the request is made to the path `/api/api/info?type=details`, because of the RequestMapping prefix path.

7. You are designing a Spring Data JPA application with entities representing 'Product' and 'Category'. A 'Product' belongs to one 'Category', and a 'Category' can have multiple 'Product's. You need to implement a method to retrieve all products within a specific category, including products from any subcategories. Which approach is the **most efficient** and adheres best to Spring Data JPA best practices for handling potentially complex category hierarchies (which may include cycles)?

Answer Choices:

(a) Add a method to the 'ProductRepository' that uses a JPQL query with a recursive CTE (Common Table Expression) to traverse the category hierarchy.

(b) Add a method to the 'CategoryRepository' that fetches the category and its subcategories (if any) and then iterates through them to retrieve the associated products using the 'ProductRepository'.

(c) Add a method to the 'ProductRepository' with a derived query using method naming conventions that attempts to recursively traverse the category hierarchy.

(d) Implement a custom query method in the 'ProductRepository' that uses multiple queries: one to fetch the category and its subcategories, and another to fetch the products for each category.

Solution key

Correct Answer: a)

Explanation:

The most efficient and scalable approach for handling complex category hierarchies, especially those that might contain cycles, is to use a recursive CTE (Common Table Expression) within a JPQL query. This approach performs the entire hierarchy traversal within the database, minimizing the amount of data transferred and the number of round trips.

Here's why the other options are less suitable:

(a) *Correct.* Recursive CTEs are designed for this type of hierarchical data traversal in the database.

(b) Incorrect. Fetching the entire category hierarchy and then iterating in Java is inefficient. It brings a lot of data into the application and performs the traversal in memory, which is slower and less scalable. It also doesn't handle cycles well.

(c) Incorrect. Derived queries based on method naming conventions are not capable of handling recursive relationships. They are designed for simpler queries based on direct properties of the entity.

(d) Incorrect. Using multiple queries is better than fetching the entire hierarchy and iterating in Java, but it's still less efficient than a single recursive CTE query. It requires multiple round trips to the database.

8. You are developing an application for a university's course catalog. You have an entity called 'Course'. You want to be able to easily find courses by their name. Which is the **most appropriate** way to add this functionality using Spring Data JPA, following its best practices?

Answer Choices:

(a) Create a custom method in the 'Course' entity that uses a JPQL query to find courses by name.

(b) Create a 'CourseRepository' interface that extends 'JpaRepository' and add a method declaration `List<Course> findByName(String name);` to it.

(c) Create a 'CourseService' class that uses the 'EntityManager' directly to query for courses by name.

(d) Create a 'CourseRepository' interface that extends 'JpaRepository' and add a method declaration `List<Course> searchCourses(String name);` to it, and then implement the method using a JPQL query in the implementation class.

Solution key

Correct Answer: b)

Explanation:

Spring Data JPA provides **query derivation** through method naming conventions in a repository interface. By declaring a method like `List<Course> findByName(String name);` in an interface that extends 'JpaRepository', Spring Data JPA automatically generates the necessary query implementation. This minimizes boilerplate code and is the recommended approach for standard queries.

(a) Incorrect. While a custom method could be added to the entity, this is generally not recommended. Entities should primarily represent data, not data access logic. Repository interfaces are the correct place for query methods.

(b) *Correct.*

(c) Incorrect. Using the 'EntityManager' directly bypasses the benefits of Spring Data JPA's repository abstraction and query derivation. It's more verbose and less maintainable for simple queries. While 'EntityManager' is necessary for more complex or dynamic queries, it is not needed for this basic use case.

(d) Incorrect. Implementing the query in the repository class defeats the purpose of Spring Data JPA's query derivation. The method naming convention approach is cleaner, more concise, and easier to maintain. While custom implementation is possible, it should be reserved for queries that can't be derived automatically.

9. Consider the following React component:

```
import React, { useRef } from 'react';

function MyComponent() {
  const myDivRef = useRef(null);

  const handleClick = () => {
    // Some DOM manipulation code will go here
  };

  return (
    <div ref={myDivRef}>
      This is a div.
      <button onClick={handleClick}>Click me</button>
    </div>
  );
}

export default MyComponent;
```

Which of the following code snippets, when placed inside the 'handleClick' function, will correctly change the background color of the div to red?

Answer Choices:

(a) `myDivRef.current.style.backgroundColor = "red";`

(b) `myDivRef.style.backgroundColor = "red";`

(c) `document.getElementById("myDivRef").style.backgroundColor = "red";`

(d) `myDivRef.current.backgroundColor = "red";`

Solution key

Correct Answer: a)

Explanation:

In React, refs are used to directly access DOM elements. The 'useRef' hook creates a mutable ref object. The '.current' property of this ref object holds the actual DOM element. Therefore, to access the div's style, 'myDivRef.current.style' is used, where the 'backgroundColor' property can be set.

(a) *Correct.*

(b) Incorrect. 'myDivRef' itself is the ref object, not the DOM element. It doesn't have a 'style' property. 'myDivRef.current' gives the the DOM element.

(c) Incorrect. 'document.getElementById' can be used if 'id' attribute set on the div, this is not the React way. Refs are the preferred method for directly accessing DOM elements in React. Also, in this example, there is no 'id' attribute.

(d) Incorrect. 'backgroundColor' is a property of the 'style' object, not directly of the DOM element itself. 'myDivRef.current.style.backgroundColor' is needed.

10. Consider the following React component:

```
import React, { useState, useEffect } from 'react';

function MyComponent() {
  const [data, setData] = useState(null);

  useEffect(() => {
    async function fetchData() {
      console.log("Before fetch");
      const response = await fetch('https://jsonplaceholder.typicode.com/todos/1'); // API call
      console.log("After fetch");
      const json = await response.json();
      console.log("After json()");
      setData(json);
      console.log("After setData");
    }

    fetchData();
    console.log("After fetchData()");

  }, []);

  if (data) {
    return <p>Data: {data.title}</p>;
  } else {
    return <p>Loading...</p>;
  }
}

export default MyComponent;
```

Which of the following sequences of console log messages is the *most likely* output when this component is rendered for the first time? (Assume the API call is successful and is reasonably fast from an external server.)

Answer Choices:

(a) `Before fetch, After fetch, After json(), After setData, After fetchData()`

(b) `Before fetch, After fetchData(), After fetch, After json(), After setData`

(c) `Before fetch, After fetchData(), After fetch, After json(), After setData, Data: ...`

(d) `Before fetch, After fetch, After json(), After fetchData(), After setData`

Solution key

Correct Answer: b)

Explanation:

Asynchronous operations and how they interact with React's 'useEffect' hook. 'await' pauses the execution of the 'async' function, but it *doesn't block* the rest of the code outside the 'async' function.

Here's the sequence of events:

(a) "Before fetch" is logged.

(b) 'fetch' is called, but the execution *pauses* at the 'await' keyword. The rest of the code in the 'useEffect' hook continues.

(c) "After fetchData()" is logged. This happens *before* the 'fetch' call completes.

(d) The 'fetch' call completes, and "After fetch" is logged.

(e) 'response.json()' is called, and the execution pauses at the 'await' keyword.

(f) "After json()" is logged.

(g) 'setData' is called, and "After setData" is logged.

The 'setData' call will trigger a re-render, but that's a separate process and doesn't affect the initial console log sequence.

(a) Incorrect. "After fetchData()" is logged *before* the 'fetch' call completes.

(b) *Correct.*

(c) Incorrect. The rendering of "Data: ..." happens *after* the initial 'useEffect' execution.

(d) Incorrect. "After fetchData()" is logged *before* "After fetch".

11. You have a React app with a piece of data (like a user's name) that needs to be used in many different parts of your app. What's the best way to make this data easily accessible to all the components that need it?

Answer Choices:

(a) Pass the data as props from parent components to child components, even if it has to go through many levels.

(b) Store the data in a central place that all components can access directly.

(c) Create a separate component just to hold the data and then render that component in all the places it's needed.

(d) Make the data a global variable so that any part of the code can access it.

Solution key

Correct Answer: b)

Explanation:

The best way to share data across many components in React is to store it in a central location, like a "shared data store." This way, any component that needs the data can access it directly, without having to pass it through many other components. This is what Context API do.

(a) Incorrect. Passing data through many levels (called "prop drilling") becomes very messy and hard to manage as the app grows.

(b) *Correct.*

(c) Incorrect. Creating a separate component just to hold the data is not the best approach. It's still a form of prop drilling, and it doesn't really solve the problem of making the data easily accessible.

(d) Incorrect. Global variables are generally bad practice. They can lead to unexpected side effects and make the code harder to maintain. React has better ways to manage shared data.

12. Consider the following React function inside a component:

```
const fetchRecordDetails = (recordId) => {
  axios.get('/records/details/' + recordId)
    .then((response) => {
        const { data } = response;
        updateHistory(data);
        return recordHistory;
    }, (error) => {
        console.log(error);
        setErrorState(<Alert severity="error">Failed to retrieve records!</Alert>);
        return null;
    });
}
```

What is the most likely output when calling 'fetchRecordDetails(456)' immediately after execution?

(a) The function returns an array containing the retrieved record history for the given ID.

(b) The function logs an error message and updates the 'errorState' if the API request fails.

(c) The function returns 'undefined' because 'updateHistory(data)' is asynchronous.

(d) The function throws an error due to an invalid state update inside 'updateHistory(data)'.

Solution key

(a) Incorrect: 'updateHistory(data)' updates the state asynchronously. The return statement immediately after 'updateHistory(data)' does not wait for the state update, so it does not return the updated record history.

(b) Correct: If the API request fails, the 'catch' block logs an error and updates 'errorState' with an error alert.

(c) Correct: Since 'updateHistory(data)' does not block execution and does not return the updated state immediately, calling 'fetchRecordDetails(456)' will return 'undefined'.

(d) Incorrect: The function does not throw an error because 'updateHistory(data)' is a valid state update inside a React component.