<Cre8tive Devs.

Software/>

MOBILE APPS•WEBSITES•AR•VIDEO GAMES•SECURITY

JVM :

Class loader :

In our AutoHire project, class loaders play a vital role in loading essential classes into the Java Virtual Machine (JVM). For example, the Customer class contains attributes such as email, first name, and last name. The class loader ensures that these customer details are properly loaded and available for our program to handle customer information effectively.

Similarly, the Car class includes attributes like ID, model, years, and manufacturer. The class loader takes the responsibility of loading these car details into the JVM, enabling our program to seamlessly manage car rentals.

By leveraging class loaders, our AutoHire project ensures that the necessary classes, along with their specific attributes, are loaded into the JVM. This allows our program to access and utilize customer and car information accurately and efficiently.

Method Area :

```java
@GetMapping
public ResponseEntity<List<CustomerResponseDto>> getAllCustomers() {
    List<CustomerResponseDto> customers = customerService.getAllCustomers();
    return new ResponseEntity<>(customers, HttpStatus.OK);
}
```

In our AutoHire project, the getAllCustomers function retrieves a list of customers from our car rental system. When this function is called, the Java Virtual Machine (JVM) allocates a specific area in the Method Area to store the bytecode instructions and metadata associated with getAllCustomers. This allows the JVM to efficiently execute the function whenever it is invoked, ensuring that the necessary instructions and data are readily available. The retrieved list of

customers is encapsulated in a ResponseEntity and returned with the appropriate HTTP status code, HttpStatus.OK.

Heap Area :

The Heap Area in the Java Virtual Machine (JVM) is a crucial memory region responsible for dynamically allocating and deallocating memory for objects in our AutoHire project. When we create objects like CustomerResponseDto, the JVM allocates memory for them in the Heap Area. The Heap Area is designed to grow and shrink as needed, allowing for efficient memory management during runtime.
In our AutoHire project, the Heap Area plays a vital role in storing and managing customer-related objects. As we retrieve customer data and create instances of CustomerResponseDto, the Heap Area accommodates these objects. The JVM's garbage collector periodically identifies and deallocates memory for objects that are no longer in use, optimizing memory utilization. The Heap Area enables our AutoHire project to handle a flexible number of objects. It allows us to dynamically manage customer data and perform operations on customer objects efficiently. However, it's important to note that the size of the Heap Area can be adjusted through JVM configuration parameters to meet the memory requirements of our application. Efficient utilization of the Heap Area is crucial to prevent memory issues like OutOfMemoryError. Therefore, it's important to design our AutoHire project with proper memory management practices, including avoiding unnecessary object creation and ensuring appropriate memory deallocation.

JIT Complier :

In the Java Virtual Machine (JVM), Just-In-Time (JIT) compilation plays a crucial role in optimizing the performance of our AutoHire project. JIT compilation is a dynamic compilation technique that transforms parts of our program's bytecode into native machine code at runtime. When our program is executed, the JVM initially interprets the bytecode. However, the JIT compiler detects frequently executed sections of code, known as "hotspots," and dynamically compiles them into native machine code. This compiled code is then executed directly by the CPU, resulting in improved performance.
In our AutoHire project, the JIT compiler identifies and optimizes critical sections, such as frequently used methods or loops, allowing them to execute more efficiently.
By converting bytecode into native machine code, JIT compilation reduces interpretation overhead, increases execution speed, and enhances overall performance.
Furthermore, the JIT compiler employs various optimization techniques, such as method inlining, dead code elimination, and loop unrolling, to further improve performance. These optimizations ensure that our AutoHire project can run faster and utilize system resources more efficiently. It's important to note that the JIT compilation process occurs dynamically during runtime, adapting to the program's execution behavior. The JVM continuously analyzes code execution patterns to identify hotspots and applies JIT compilation accordingly.
Overall, JIT compilation in the JVM enhances the performance of our AutoHire project by dynamically converting frequently executed bytecode into optimized native machine code. It enables faster execution, reduced interpretation overhead, and efficient utilization of system resources, ultimately improving the overall responsiveness and efficiency of our application.

Garbage Collector :

In our AutoHire project, the Garbage Collector in the Java Virtual Machine (JVM) plays a vital role in managing memory, including the objects created and used within the addCar function. As the addCar function executes, various objects are created and referenced, such as the CarRequestDto, CarResponseDto, and the ResponseEntity. The Garbage Collector constantly monitors these objects and identifies those that are no longer needed or referenced. In the case of the addCar function, once the CarResponseDto object is created and returned as part of the ResponseEntity, the Garbage Collector recognizes that the CarResponseDto object is no longer actively used or referenced outside of the function. Therefore, it marks the CarResponseDto object as eligible for garbage collection.

When the Garbage Collector runs, it detects the unused CarResponseDto object and reclaims the memory it occupies. This process ensures efficient memory utilization within the JVM and prevents memory leaks, freeing up resources for future object allocations. By automatically managing memory deallocation, the Garbage Collector allows us to focus on writing code logic without explicitly worrying about freeing memory. It enhances the stability and performance of our AutoHire project, ensuring that unused objects are efficiently removed from memory, thus preventing memory-related issues.

Testing (Debugging and Integration Testing) :

Breakpoint :
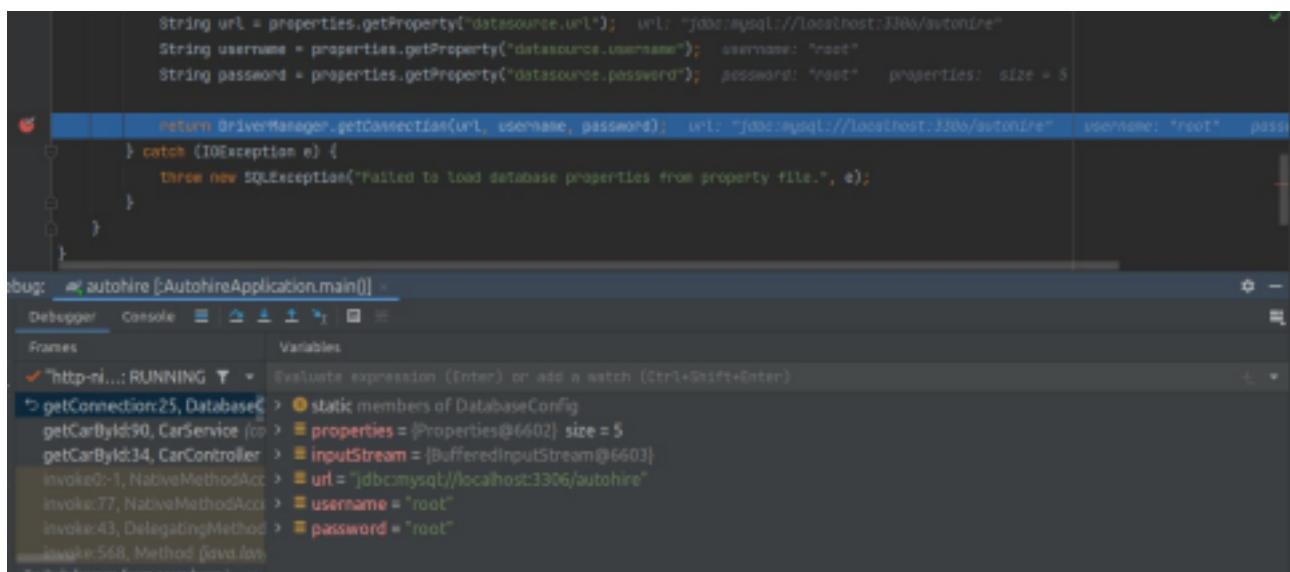
Breakpoints are markers we can set in our code during debugging to pause the execution at a specific line or method. They provide a way to inspect variable values, analyze the program's behavior, and troubleshoot issues.

By setting a breakpoint at the return statement in the "getConnection" method, we could pause the code execution just before the connection was returned. This allowed us to inspect and verify the values of variables like "url," "username," and "password" before establishing the database connection.
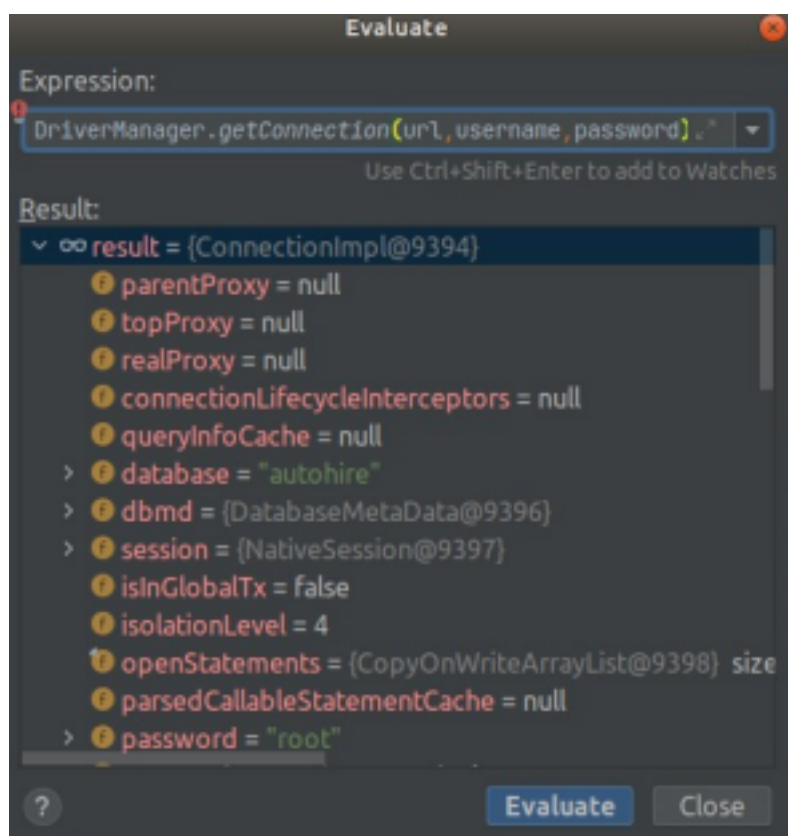The benefits of setting a breakpoint at the return statement included:
1. Variable inspection: We could examine the values of these variables, ensuring their correctness and identifying any potential issues in the retrieved properties.
2. Troubleshooting: If there were any errors or unexpected behavior, the breakpoint allowed us to analyze the state of variables and pinpoint the source of the problem, facilitating effective troubleshooting.
3. Verification: By leveraging the breakpoint, we could validate that the properties were successfully loaded from the property file and used correctly to establish the database connection. This ensured the accuracy and reliability of the connection details.
Setting a breakpoint at the return statement provided us with greater control and insight during debugging, allowing us to validate the connection setup and address any issues before the connection was returned to the caller.

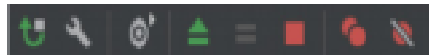**Evaluator, Debugger, and Stepping and Play Button** :

Evaluator: The Evaluator allows you to manually evaluate expressions or variables during runtime. With the Evaluator, you can inspect the values of variables like "url," "username," and "password" just before the connection is returned. It provides a way to verify if the retrieved property values are correct and aligned with your expectations.

Debugger: The Debugger is a powerful tool that helps you find and fix issues in your code. By setting a breakpoint at the return statement in this function, you can activate the Debugger. It allows you to control the execution flow and observe the state of variables at that specific point. The Debugger lets you pause the code execution, examine variable values, step through the code, and understand the behavior of your program. It assists in identifying any potential issues in retrieving and using the database connection details.

Stepping and Play Buttons: The Stepping buttons, such as Step Over, Step Into, and Step Out, provide control over the execution flow while debugging. When the breakpoint is hit, you can use the Stepping buttons to navigate through the code line by line. The Step Over button enables you to execute the current line and move to the next line, allowing you to observe the variable assignments and the flow of execution within the try-catch block. The Play button is used to resume the program's normal execution after the breakpoint, allowing the return statement to be executed and the connection to be returned.

JDBC (Using Java Advance) :

Obtaining Database Connection from Property File :

In Java, a database connection allows a Java application to interact with a database management system (DBMS), such as MySQL. It enables the application to perform various operations like querying, updating, and manipulating data in the database.
To establish a database connection in Java, several essential components are involved. These include the database driver, connection URL, authentication credentials (username and password), and optional additional properties.

The process typically involves the following steps:

    1. Loading the driver: Before establishing a connection, the appropriate database

driver needs to be loaded. This is achieved by including the driver's JAR file in the classpath or using a built-in driver manager.

2. Creating a connection URL: The connection URL specifies the location and details of the database. It typically includes information such as the database type, hostname, port, and database name. In our case, the connection URL is "jdbc:mysql://localhost:3306/autohire", indicating a MySQL database running on the local machine with the "autohire" database name.

3. Providing authentication credentials: The username and password are required to authenticate the application's access to the database. In our case, the username is "root" and the password is "root".

4. Establishing the connection: Using the driver manager and the connection URL, a connection object is created. This connection object represents a live connection to the database and enables the execution of SQL statements and retrieval of results.

Now, let's illustrate the process using the provided method, "getConnection":

The "getConnection" method follows the standard procedure for establishing a database connection in Java. It loads the necessary properties from a property file, which contains the required connection details such as the URL, username, and password.

The loaded properties include:

- datasource.url: "jdbc:mysql://localhost:3306/autohire"

- datasource.username: "root"

- datasource.password: "root"

The method utilizes the "Properties" class to hold the loaded properties. It then retrieves the connection details from the properties object.

Finally, the method uses the DriverManager class, a part of the Java SQL API, to establish the database connection. It passes the connection URL, username, and password obtained from the properties to the getConnection method, which returns a connection object representing the established connection.

By utilizing the "getConnection" method with the provided properties, we successfully obtain a database connection in the described manner. The loaded properties provide the necessary details
for connecting to the MySQL database running on the local machine with the "autohire" database, and the DriverManager facilitates the actual connection establishment.

Connection Pooling :

Connection pooling in Java is an advanced technique that manages and reuses database connections efficiently. It optimizes the process of establishing and closing connections, leading to improved performance, scalability, and resource utilization. By reusing connections from a pool rather than creating new ones for each request, connection pooling minimizes overhead and enhances application responsiveness. Advanced features include connection reuse, pool sizing, connection validation, leak detection, and timeout/recovery. Connection pooling improves performance, scalability, resource

utilization, and stability in database interactions.

Transaction Management :

Transaction management refers to the process of ensuring data integrity and consistency when executing multiple database operations as a single, indivisible unit of work. It guarantees that either all the operations within a transaction are successfully completed, or none of them are applied to the database. By managing transactions, you can maintain the integrity of your data and handle exceptions effectively, ensuring that your database remains in a valid and consistent state.

```java
public CarResponseDto addCar(CarRequestDto carRequestDto) {

    String insertQuery = "INSERT INTO car (manufacturer, model, year, color, price) VALUES (?, ?, ?, ?,?)";

    try (Connection connection = DatabaseConfig.getConnection();
         PreparedStatement preparedStatement = connection.prepareStatement(insertQuery, PreparedStatement.RETURN_GENERATED_KEYS)) {

        preparedStatement.setString( parameterIndex 1, carRequestDto.getManufacturer());
        preparedStatement.setString( parameterIndex 2, carRequestDto.getModel());
        preparedStatement.setInt( parameterIndex 3, carRequestDto.getYear());
        preparedStatement.setString( parameterIndex 4, carRequestDto.getColor());
        preparedStatement.setLong( parameterIndex 5,carRequestDto.getPrice());

        int rowsAffected = preparedStatement.executeUpdate();

        if (rowsAffected == 1) {
            ResultSet generatedKeys = preparedStatement.getGeneratedKeys();
            Long carId = null;
            if (generatedKeys.next()) {
                carId = generatedKeys.getLong( columnIndex 1);
            }
        }
```

In the given method, we performed transaction management to ensure data integrity and consistency when adding a new car to the database using JDBC. Let's delve into the details:

1. Database Operations: We executed a database operation to add a new car by preparing an insert query, setting the necessary parameters, and executing the query using JDBC.
2. Transaction Management: We explicitly managed the transaction by disabling the auto commit mode in JDBC. By default, JDBC operates in auto-commit mode, where each SQL statement is treated as a separate transaction and automatically committed to the database. However, in our method, we disabled auto-commit mode to encompass multiple SQL statements within a single transaction.

3. Disabling Auto-commit: We ensured that the subsequent SQL statements executed within the same connection formed part of the ongoing transaction. This allowed us to control the transaction boundaries manually.

4. Committing the Transaction: After executing the insert query and confirming its success, Java committed the transaction explicitly using connection.commit(). This finalized the transaction and made the changes permanent in the database.

5. Exception Handling and Rollback: In case of any SQLException during the database operation, Catch block caught the exception and performed a rollback

using connection.rollback(). This ensured that any changes made within the transaction were discarded, maintaining data consistency and integrity.

By explicitly managing the transaction in this method, we ensured that the insertion of a new car was treated as an atomic unit of work. If any part of the operation failed, the entire transaction was rolled back, preventing partial changes from being persisted in the database.

Basics of Multithreading :

Functions of Thread Class :

Threads in Java represent lightweight units of execution that allow concurrent programming. They enable multiple tasks to run simultaneously within a single program, each executing independently of the others.

```java
Runnable emailTask = () -> sendEmail(email, subject, body, receiptFileName);
Thread emailThread = new Thread(emailTask);
emailThread.start();
```

In the provided code snippet, we implemented a thread using the Runnable interface and the Thread class. Here's how we did it:

1. Runnable Interface: We created a Runnable implementation using a lambda expression. The Runnable represents the task we want to execute concurrently, which in this case is sending an email with specific details.

2. Task Definition: Inside the lambda expression, we defined the sendEmail method with the necessary parameters: email, subject, body, and receiptFileName. This method contains the logic to send an email.

3. Thread Creation: We instantiated a new Thread object, emailThread, and passed the emailTask as a parameter to its constructor. The emailTask represents the task we want to execute concurrently in a separate thread.

4. Thread Start: We invoked the start() method on the emailThread to start the execution of the task in a new thread. The start() method internally calls the run() method defined in the Runnable interface.

By creating a new thread and executing the sendEmail task asynchronously, we allow the email to be sent concurrently with other operations in the program. This promotes parallelism and can enhance the responsiveness and performance of the application.

It's important to note that in this implementation, the Runnable interface and the Thread class were used to create and manage the concurrent execution of the sendEmail task. The Runnable interface provided a way to define the task, and the Thread class facilitated the creation and execution of a new thread to run the task asynchronously.

Intercommunication between Threads :

Thread communication in Java involves the coordination and exchange of information between multiple threads to achieve synchronization and orderly execution. It allows threads to interact, share data, and coordinate their actions to achieve desired outcomes.

1. wait() Method: The wait() method is used by a thread to pause its execution and wait for a specific condition to be met. For example, the shopkeeper thread may wait until a desired item is restocked before serving the customer. This method ensures that the thread does not proceed until the condition is fulfilled.

2. notify() Method: The notify() method is used to notify a waiting thread that a specific condition has been met. In the context of our example, the customer thread uses notify() to inform the shopkeeper thread that they are ready to make a purchase. This allows the shopkeeper to proceed with serving the customer.

3. synchronized Blocks: Synchronization using the synchronized keyword helps ensure safe interaction between threads. It allows threads to access shared resources, such as the inventory or payment system, in a controlled manner. This prevents conflicts and maintains data consistency.

In our scenario, the shopkeeper thread waits using wait() until the desired item is available. The customer thread then notifies the shopkeeper using notify(), indicating their readiness to make a purchase. Synchronized blocks ensure that shared resources are accessed safely by both threads.

Synchronized Methods and Blocks :

Synchronized methods and blocks in Java enable controlled access to shared resources in a ticketing system. For example, when multiple users try to book tickets simultaneously, synchronized methods can be used to handle the booking process. Inside the synchronized block, critical sections of code
that update shared resources, such as the seating availability or payment
system, can be encapsulated.

In the ticketing system, a synchronized block can be utilized to ensure that only one thread at a time can access and modify the seating availability data. This prevents conflicts where multiple users might select the same seat simultaneously. By acquiring a lock associated with the shared resource, each thread takes turns executing the synchronized block, maintaining the integrity of the seating availability and avoiding double bookings.

Synchronized methods and blocks provide a way to synchronize access to shared resources, ensuring data consistency and avoiding conflicts in the ticketing system. By properly implementing synchronization, the ticketing system achieves concurrent handling of ticket bookings while maintaining a reliable and accurate representation of available seats.

Lambda Expressions and Streams :

Parallel Streams :

Parallel streams in Java allow for concurrent processing of collections or arrays. They leverage multiple threads to improve performance on large datasets by dividing the workload into smaller chunks that are processed concurrently. This parallel execution can lead to faster computations, especially on multi-core processors. Parallel streams are created using the .parallelStream() method and offer a convenient way to leverage parallelism in your code.

```
customer.getCarList().parallelStream()
        .forEach(car -> totalPrice.add(car.getPrice()));
```

The *parallelStream()* method is invoked on the *carList* of the *customer* object. This creates a parallel stream for concurrent processing.

By using parallelStream(), the car list is divided into smaller chunks, and each chunk is processed concurrently by separate threads. This enables parallel execution of the subsequent operations.

The forEach operation applies the provided action to each element in the parallel stream. In this case, the lambda expression car -> totalPrice.add(car.getPrice()) is applied to each car object. The lambda expression adds the price of the car to the totalPrice variable.

Methods of Streams & Functional Interface :

Streams in Java provide a functional and efficient way to process collections of data. They allow you to perform operations on data sets in a declarative and expressive manner, enabling efficient and concise code.

 Filter Method in Streams:

The filter method is an intermediate operation in a stream pipeline. It is used to select elements from a stream based on a given condition. It takes a Predicate functional interface as a parameter, which defines the condition for filtering. The filter method returns a new stream that contains only the elements that satisfy the specified condition.

```
cars = cars.stream().filter(carResponseDto -> carResponseDto.getIsBooked() == false).collect(Collectors.toList());
```

In this example, the filter method is applied to the cars stream. It uses a lambda expression carResponseDto -> carResponseDto.getIsBooked() == false as the predicate. The condition checks whether the isBooked property of each CarResponseDto object is false. Only the elements that satisfy this condition will be included in the resulting stream.

After filtering, the collect method with Collectors.toList() is used to collect the filtered elements into a new list.

By utilizing the filter method, we effectively select and retain only the cars that are not booked from the original cars stream. This allows us to work with a filtered dataset and perform further operations or actions specifically on those elements.

The use of the filter method simplifies the filtering process, providing a concise and readable way to select elements based on specific conditions in a stream.

Functional interfaces in Java are interfaces that contain only one abstract method. They play a crucial role in functional programming, as they can be used as the basis for lambda expressions or method references.

```
Runnable emailTask = () -> sendEmail(email, subject, body, receiptFileName);
Thread emailThread = new Thread(emailTask);
emailThread.start();
```

In our project, we utilized the Runnable functional interface for handling asynchronous tasks. We created a Runnable object using a lambda expression and executed it in a separate thread using the Thread class. This allowed us to perform tasks concurrently, enhancing the responsiveness of our application. Functional interfaces like Runnable provide a concise and expressive way to handle tasks in a functional programming style.

Collections :

Data Structures and Related Collections :

Data structures in Java refer to the way data is organized and stored in memory. Java provides a variety of data structures, and related collections, which are implemented through classes and interfaces in the Java Collections Framework. These collections offer different ways to store, manipulate, and access data, based on the specific requirements of your application.

1. Set: A Set is a collection that stores unique elements. It ensures that each element is unique by using methods like add and contains. Common implementations include HashSet and TreeSet.

2. List: A List is an ordered collection that allows duplicate elements. It provides

methods for accessing, adding, and removing elements by index. Common implementations include ArrayList and LinkedList.

3. ArrayList: ArrayList is an implementation of the List interface that dynamically grows as elements are added. It provides random access to elements using indexes and is efficient for retrieving and modifying elements.

```java
public CustomerResponseDto getCustomerById(Long customerId) {
    String selectQuery = "SELECT * FROM customer WHERE id = ?";
    String carCustomerQuery = "SELECT car_id FROM car_customer where customer_id=?";
    String selectCarQuery = "SELECT * FROM car WHERE id = ?";
    try (Connection connection = DatabaseConfig.getConnection();
         PreparedStatement preparedStatement = connection.prepareStatement(selectQuery)) {
        PreparedStatement preparedStatementCarQuery = connection.prepareStatement(carCustomerQuery);
        PreparedStatement carSelectStatementQuery = connection.prepareStatement(selectCarQuery);
        preparedStatementCarQuery.setLong( parameterIndex 1,customerId);
        ResultSet resultSetCarId = preparedStatementCarQuery.executeQuery();
        Set<Long> carIds = new HashSet<>();
        List<CarResponseDto> carResponseDtos = new ArrayList<>();
        while(resultSetCarId.next()){
            Long carId = resultSetCarId.getLong( columnLabel: "car_id");
            carIds.add(carId);
        }
```

First, we employed a HashSet to store the car IDs retrieved from the database. This collection ensured that each ID was unique, preventing duplicates.

Next, we used an ArrayList to maintain a list of CarResponseDto objects. This list provided an ordered collection, allowing us to access, add, and remove elements based on their index positions.

By utilizing the ArrayList implementation, we benefited from its dynamic resizing capability, which allowed the collection to grow as elements were added.

Throughout the code, we accessed the collections using appropriate methods such as add(), contains(), and get(), to perform operations on the stored data. These operations included retrieving car IDs, fetching corresponding car details, and constructing CarResponseDto objects.

Overall, these collections provided us with a structured approach to storing and managing data, ensuring efficient retrieval and manipulation.
Map and Concurrent Map :

Map is an interface that represents a collection of key-value pairs.

It allows you to associate values with unique keys and perform operations based on those

keys. Common implementations include HashMap, TreeMap, and LinkedHashMap.

ConcurrentMap is a sub-interface of Map that provides thread-safe operations for

concurrent access. It extends Map and adds additional atomic and thread-safe operations.

ConcurrentMap implementations, like ConcurrentHashMap, handle concurrent access from multiple threads.

They ensure thread-safety and high performance in concurrent

environments. Maps are commonly used for caching, indexing, and

fast lookup operations.

Generics :

Need of Generics and Collections with Generics :

Generics in Java provide a way to define classes, interfaces, and methods that can operate on various types of objects while maintaining type safety. Generics allow you to parameterize classes and interfaces with type arguments, enabling you to create reusable and type-safe code.

In the given code snippet:

```java
List<CarResponseDto> carResponseDtos = new ArrayList<>();
```

We utilized generics by specifying the type argument <CarResponseDto> when declaring the carResponseDtos list. This ensured that the list would only accept objects of type CarResponseDto.

Advantages of using generics:

1. Type Safety: Generics provide compile-time type checking, allowing for early detection of type-related errors. By using generics, we ensured that the carResponseDtos list would only contain CarResponseDto objects, preventing type mismatches at runtime.

2. Code Reusability: With generics, we created a generic list that could be used to store objects of a specific type. This promotes code reusability, as the same list implementation can be used with different types by simply changing the type argument.

3. Enhanced Readability: By explicitly specifying the type argument, we made the code more self-explanatory and easier to understand. It conveyed the intended usage of the list and provided clarity to other developers who might work with the code.

4. Avoiding Type Casting: Generics eliminate the need for explicit type casting, reducing the risk of runtime errors. In our case, since we specified the type argument as
    CarResponseDto, we could directly retrieve elements from the list without the need for casting.

By incorporating generics in the code, we achieved type safety, improved code reusability, enhanced readability, and reduced the chances of type-related errors, resulting in more robust and maintainable code.

Wildcards :

The use of wildcards in Java allows for more flexibility and generality in dealing with generic types. In the provided code snippet from the autohire project:

```java
@PostMapping(value = "addcar")
public ResponseEntity<? extends AbstractCarDto> addCar(@RequestBody CarRequestDto carRequestDto)
    if (!CarRequestValidator.isValidCarRequestDto(carRequestDto)) {
        throw new InvalidArgumentException("Invalid car details");
    }
    CarResponseDto createdCar = carService.addCar(carRequestDto);
    return new ResponseEntity<>(createdCar, HttpStatus.CREATED);
}
```

We utilized the wildcard ? extends AbstractCarDto in the return type of the addCar method. This wildcard represents an unknown subtype of AbstractCarDto. It allows the method to return any subtype of AbstractCarDto, providing flexibility in the response entity.

Advantages of using wildcards:

1. Flexibility: By using the wildcard ? extends AbstractCarDto, the method can return different subclasses of AbstractCarDto. This enables the method to handle a wider range of concrete CarResponseDto types without explicitly specifying each subtype.
2. Reusability: The wildcard allows the addCar method to be used with various subclasses of AbstractCarDto without modifying the method's signature. This promotes code reusability, as the same method can handle different types of car response DTOs.
3. Compatibility: The use of a wildcard allows the method to be compatible with future subclasses of AbstractCarDto that may be added to the project. This ensures that the method remains adaptable to changes in the codebase.

In the context of the autohire project, using the wildcard in the addCar method allows for a more flexible and generic approach to returning different types of AbstractCarDto subclasses as the response entity. It enhances code reusability and ensures compatibility with potential future subclasses.

In the autohire project, we utilized the upper bounded wildcard in the following code snippet:

By using ? extends AbstractCarDto, we specified that the response entity can be any subclass of AbstractCarDto or AbstractCarDto itself. This allowed for flexibility in returning different types of car response DTOs.

For example, if we have subclasses such as CarResponseDto and CarRequestDto that

extend AbstractCarDto, the addCar method could return instances of these subclasses. The upper bounded wildcard enabled the method to handle various concrete types that extend AbstractCarDto.

This approach provided greater flexibility in the response entity, as it allowed for different types of car response DTOs to be returned without explicitly specifying each subtype. It promoted code reusability, as the same method could handle different concrete types that extend the base AbstractCarDto class.

12 factory App :

In our project, we implemented the 12 Factory App pattern by utilizing an Abstract CarDto as the common interface or abstract class for the products. This Abstract CarDto defined the common behavior that all car-related DTOs should implement.

One of the concrete classes we created was CarResponseDto, which represented the response data transfer object for a car entity. It extended the Abstract CarDto and provided specific implementation details relevant to representing car data in a response.

 Similarly, we also implemented CarRequestDto as another concrete class that extended the Abstract CarDto. This class encapsulated the data needed to request the creation or modification of a car entity.

By utilizing the 12 Factory App pattern, we achieved a modular and extensible design for managing car-related data transfer objects. The Abstract CarDto acted as the contract defining the common behavior, while the CarResponseDto and CarRequestDto served as concrete implementations tailored for specific use cases.

This approach allowed us to decouple the creation and usage of car-related DTOs, promoting code reuse and maintainability. It provided a clear separation of concerns, enabling us to handle different
types of car-related data efficiently and in a consistent manner throughout our project.

Reflection :

Accessing Fields Dynamically & Validation Using Reflection :

Reflection in Java is a powerful feature that allows a program to examine or modify its own structure at runtime. It provides the ability to inspect and manipulate classes, interfaces, fields, methods, and constructors dynamically.

```java
public static boolean isValidCarRequestDto(CarRequestDto dto) throws IllegalAccessException {
    Field[] fields = dto.getClass().getDeclaredFields();

    for (Field field : fields) {
        field.setAccessible(true);

        // Check if field is manufacturer or model
        if (field.getName().equals("manufacturer") || field.getName().equals("model")) {
            String value = (String) field.get(dto);
            if (value == null || value.isEmpty()) {
                return false;
            }
        }

        // Check if field is year
        if (field.getName().equals("year")) {
            int value = field.getInt(dto);
            if (value == 0) {
                return false;
            }
        }
    }
```

In our implementation, we utilized reflection to validate a CarRequestDto object dynamically. Here's how the code leverages reflection and its benefits:

1. Obtaining Class Information: We used dto.getClass() to retrieve the Class object representing the runtime class of the CarRequestDto object. This allows us to access information about the class's structure, including its fields.

2. Accessing Fields: By calling getDeclaredFields() on the Class object, we obtained an array of Field objects representing all the fields declared in the CarRequestDto class. This enables us to inspect and manipulate the fields programmatically.

3. Modifying Field Accessibility: We invoked setAccessible(true) on each Field object to modify its accessibility. This allows us to access private fields that are not normally accessible. It gives us the ability to examine and modify fields regardless of their visibility modifiers.

4. Field Validation: With reflection, we iterated through the fields and performed specific validation based on their names. We checked if the field is "manufacturer" or "model" and verified if their values are null or empty. Additionally, we checked the "year" field and ensured it is not zero.

Using reflection in this context allowed us to dynamically examine and validate the fields of the CarRequestDto object without explicitly referring to field names. It provided a more flexible and generic approach to field validation, eliminating the need for hardcoded field names and reducing code duplication. Reflection facilitated the introspection and manipulation of objects, making our code more adaptable, reusable, and capable of handling dynamic scenarios.

Exception Handling :

User-defined Exceptions :

Exception handling in Java is a mechanism that allows us to handle and manage errors or exceptional conditions that may occur during program execution. It helps in maintaining program flow, providing error messages, and taking appropriate actions when exceptions occur.

In Java, exceptions are represented as objects derived from the Throwable class. There are two main types of exceptions: checked exceptions and unchecked exceptions. Checked exceptions are required to be handled explicitly using try-catch blocks or declared in the method signature using the throws keyword. Unchecked exceptions, also known as runtime exceptions, do not require explicit handling.

One common practice in exception handling is to define user-defined exceptions that extend either the Exception class or one of its subclasses. These user-defined exceptions allow us to create custom exception types that are specific to our application's requirements.

```java
if (!CustomerRequestValidator.isValidCustomerRequestDto(bookVehicleDto)) {
    throw new InvalidArgumentException("Please check , email , firstname , lastname and email shou
}
```

```java
8 usages    Akshay kamble
public class InvalidArgumentException extends RuntimeException {
    1 usage
    private String message;
    2 usages    Akshay kamble
    public InvalidArgumentException(String message) {
        super(message);
        this.message = message;
    }
}
```

In our project, we have implemented the above code snippet for validating a CustomerRequestDto object using a custom user-defined exception called InvalidArgumentException.

The code checks if the provided CustomerRequestDto is valid by invoking the static method isValidCustomerRequestDto() from the CustomerRequestValidator class. If the validation fails, indicating that the email, first name, last name, or email format is incorrect, an instance of the InvalidArgumentException is thrown.

The InvalidArgumentException is constructed with an error message describing the issue,

such as "Please check, email, first name, last name, and email should be in proper format". By throwing this exception, we indicate that an invalid argument has been provided, allowing the calling code to handle the exception appropriately.

This approach provides a clear and descriptive way to handle validation errors. By using a custom exception, we can differentiate these specific validation-related exceptions from other exceptions that may occur in the application. It also helps in communicating the reason for the exception to the caller or providing a meaningful error message to aid in troubleshooting and resolution.

File Operations :

Read and Write File in Java :

```java
String receiptFileName = "receipt_" + customerCarId + ".pdf";

try {
    ITextRenderer renderer = new ITextRenderer();

    StringBuilder htmlContent = new StringBuilder();
    htmlContent.append("<html><head><style>");
    htmlContent.append("body { font-family: Arial, sans-serif; }");
    htmlContent.append("h1 { font-size: 20px; }");
    htmlContent.append("p { font-size: 12px; }");
    htmlContent.append("</style></head><body>");
    htmlContent.append("<h1>AutoHire Car Rental - Receipt</h1>");
    htmlContent.append("<p>Manufacturer: ").append(manufacturer).append("</p>");
    htmlContent.append("<p>Model: ").append(model).append("</p>");
    htmlContent.append("<p>Price: $").append(formattedPrice).append("</p>");
    htmlContent.append("<p>Taxes: $").append(formattedTaxes).append("</p>");
    htmlContent.append("<p>Total: $").append(formattedTotal).append("</p>");
    htmlContent.append("</body></html>");

    renderer.setDocumentFromString(htmlContent.toString());

    try (OutputStream outputStream = new FileOutputStream(receiptFileName)) {
```

File operations in Java involve tasks related to files and directories, such as reading, writing, creating, deleting, and manipulating files. Java provides classes and methods in the java.io and java.nio packages to handle file operations. These operations are useful for tasks like reading configuration files, logging data, generating reports, and working with file-based resources. By utilizing Java's file operation capabilities, developers can efficiently manage file-related tasks, enhancing data processing and storage functionalities in their applications.

In our project, we have implemented the above code snippet for performing file operations in Java. Let's understand the code and its purpose:

1. File Naming: The code generates a file name for the receipt using the customerCarId. It appends "receipt_" to the car ID and adds the ".pdf" extension. This ensures a unique and identifiable file name for each receipt.

2. Creating HTML Content: The code constructs an HTML content string using a StringBuilder. It includes header and paragraph tags to structure the content of the receipt. It dynamically inserts data such as the manufacturer, model, price, taxes, and total.

3. Rendering HTML to PDF: The code uses the ITextRenderer class from the iText

library to render the HTML content as a PDF document. The setDocumentFromString() method is used to pass the HTML content to the renderer.

4. Writing the PDF: The code creates an output stream to a file using the FileOutputStream class. It then calls the layout() method on the renderer to perform layout calculations for the PDF. Finally, the createPDF() method is invoked, which writes the PDF content to the output stream, effectively creating the receipt PDF file.

By executing this code, we generate a receipt in PDF format based on the provided data. This file operation is useful for generating and saving various types of documents, such as receipts, reports, or any other formatted content, in a file format that can be easily shared, printed, or archived.
The code demonstrates the capabilities of Java for file handling, content generation, and rendering, enabling us to perform file-based operations efficiently in our project.

Maven vs Gradle :

Popular build automation technologies used in Java and other JVM-based applications include Maven and Gradle. They manage project dependencies, compile source code, run tests, and package artefacts for the same basic reason. They have different approaches and configuration methods, though.

Maven:

One of the first and most popular build tools in the Java environment is Maven.

The structure, dependencies, and build objectives are specified in a configuration file called "pom.xml" (Project Object Model) that is based on XML.

By enforcing a uniform project layout, Maven adopts a convention-over-configuration methodology, which makes it simpler for developers to comprehend the project's structure with minimal configuration.

A centralised repository system is used to handle dependencies, and Maven uses a hierarchical dependency resolution technique to acquire necessary libraries.The phases of the Maven build lifecycle, "compile," "test," "package," "install," and "deploy," are executed in that order and are predetermined. The specified phases can be bound to by developers with their unique actions.

It is a dependable option for conventional Java projects due to its reliability and resilience.

Gradle:

Maven was replaced by Gradle, a more advanced build tool that provides more flexibility and capability.

For build script setup, a Domain-Specific Language (DSL) based on Kotlin or Groovy is used. Developers are now able to create build scripts that are both shorter and more expressive.

Like Maven, Gradle favours convention over configuration, but it still supports highly

configurable build setups when necessary.

Its dependency management is adaptable and can handle a variety of repository types, including local repositories, Maven Central, and Ivy.

Since Gradle builds are based on a directed acyclic graph (DAG), they may be executed in parallel and in stages, which dramatically boosts build efficiency for complicated projects.

Developers may more effectively specify complicated projects and their interdependencies thanks to its seamless support for multi-project builds.Gradle's extensibility and community plugins ecosystem enable users to leverage a wide range of functionalities beyond the default build capabilities.
Dependecies in Maven :

You must modify the "pom.xml" file in Maven in order to add dependencies. An illustration of how to add the Apache Commons Lang library as a dependency is as follows:

Your Maven project's "pom.xml" file should be opened.

Find the dependencies> section and insert the next piece of XML there:

```xml
<dependency>
    <groupId>com.mysql</groupId>
    <artifactId>mysql-connector-j</artifactId>
    <scope>runtime</scope>
</dependency>
```

Dependecies in Gradle :

You must modify the "build.gradle" file in Gradle in order to add dependencies. An illustration of how to add the Apache Commons Lang library as a dependency is as follows:

Go to your Gradle project and open the "build.gradle" file.

Add the next line to the dependents block by finding it and locating it:

```
runtimeOnly 'com.mysql:mysql-connector-j'
```

How to avoid outOfMemory Error :

Increase the JVM Heap Size: When the JVM heap is insufficient to fulfil the memory requirements of the programme, OutOfMemoryError frequently arises. Using JVM options like -Xmx (maximum heap size) and -Xms (initial heap size), you can expand the heap size. For instance, -Xmx2G limits the size of the heap to 2 GB.

Monitor Memory Usage: Keep an eye out for any unusual patterns in your application's memory usage. JVisualVM, JConsole, or Java Mission Control are a few examples of tools you can use to examine memory use and find possible memory leaks.

Close Resources Appropriately: When you're finished using them, make sure to explicitly close all resources, including files, streams, and database connections. Resource leaks and extra memory usage might result from not terminating resources.

Use effective algorithms and data structures: To optimise memory use, pick the right data structures   and algorithms. Use StringBuilder rather than string concatenation, for instance, when manipulating lengthy strings.
Batch processing: Instead of loading the entire dataset into memory at once, take into account processing it in batches if you are working with a huge amount of data.

Instead of gathering data into lists or arrays, consider using streams in some situations to process data more efficiently with regard to memory use.

Avoid Creating Too Many Threads: Too many threads can cause the system to use more memory. Utilise thread pools or asynchronous programming to effectively manage multiple tasks at once.