ROBOT RACE DOCUMENTATION

Project Overview

The Robot Race application is a Java program that simulates a race between multiple robots on a 2D cartesian grid. Each robot is given a set of movement instructions, a delay time between each instruction, and a starting position and facing direction. The goal is to execute the instructions for each robot simultaneously and track their final positions and rankings.

Design and Implementation

The project follows an object-oriented design approach and is organized into several packages:

1. `exercise1.robot.enums`: This package contains the `FacingDirection` and `MovementInstruction` enums, which represent the robot's facing direction and the possible movement instructions, respectively.

2. `exercise1.robot.handleException`: This package contains custom exception classes (`IllegalCommand` and `IllegalFaceDirection`) and a utility class (`ValidationCommand`) for validating movement instructions.

3. `exercise1.robot.models`: This package contains the core classes for modeling the robot and the race:
   - `Coordinate`: Represents a 2D coordinate point.
   - `Robot`: Encapsulates the robot's state (current position and facing direction) and movement logic.
   - `RobotInterface`: Defines the contract for a robot in the race.
   - `RobotRace`: Extends the `Robot` class and implements the `Runnable` interface, representing a robot participating in the race. It handles the robot's movement, delay, and ranking.

4. `exercise1.robot.utils`: This package contains utility classes:
   - `CalculateCorrectDirectionUtility`: Provides methods for calculating the correct facing direction based on the current direction and movement instruction.
   - `ModelUtility`: Contains a utility method for converting a string to a list of characters.

Key Components

1. Robot Modeling:
  - The `Robot` class represents a single robot and encapsulates its state (current position and facing direction) and movement logic.
  - The `move(String instructions)` method processes the movement instructions for the robot.
  - The `moveForward()`, `turnLeftOperation()`, and `turnRightOperation()` methods handle the robot's movement based on the instructions.

2. Robot Race:
  - The `RobotRace` class extends the `Robot` class and implements the `Runnable` interface, allowing each robot to run in its own thread.
  - The `run()` method is the entry point for the robot's thread, where it executes the movement instructions with the specified delay.
  - The `rankHolder` is a static queue that stores the ranked robots.
  - The `getElapsedTime()` method returns the elapsed time since the start of the race.

3. Input Validation and Exception Handling:
  - The `ValidationCommand` class provides a utility method `isValidCommand(String commands)` to validate the input sequence of movement instructions.
  - The `IllegalCommand` and `IllegalFaceDirection` exceptions are thrown when invalid movement instructions or facing directions are encountered.

4. Utility Classes:
  - The `CalculateCorrectDirectionUtility` class handles the logic for updating the robot's facing direction based on the movement instructions.
  - The `ModelUtility` class provides a helper method for converting a string to a list of characters.

Testing

The project includes unit tests written using JUnit 4 to ensure the correctness of the implemented functionality. The tests cover various scenarios, including valid and invalid input, edge cases, and the expected behavior of the robot's movement and direction changes.

Potential Enhancements

The project is designed with extensibility in mind, making it easier to incorporate future enhancements. Some potential enhancements mentioned in the problem statement include:

1. Misguided Robot: A robot that turns or moves in the opposite direction of its commands could be implemented by introducing a new class that extends the `Robot` class and overrides the `moveForward()`, `turnLeftOperation()`, and `turnRightOperation()` methods to handle the misguided behavior.

2. Speedy Robot: A robot that moves forward three times as far as commanded could be implemented by introducing a new class that extends the `Robot` class and overrides the `moveForward()` method to handle the increased movement distance.

3. Battery-powered Robots: Robots with limited energy could be implemented by introducing a new class that extends the `Robot` class and keeps track of the robot's energy level. The movement methods could be modified to decrement the energy level with each movement, and the robot could stop executing commands when the energy level reaches a certain threshold.

Overall, the Robot Race application demonstrates a solid understanding of **object-oriented programming principles, including encapsulation, inheritance, and polymorphism.** The code is well-organized, follows best practices, and **includes input validation and exception handling mechanisms to ensure robustness.** The modular design and the use of interfaces facilitate future extensibility and maintainability.

Different Packages

Enums

1. `FacingDirection.java`:
   - This enum represents the four cardinal directions: NORTH, SOUTH, EAST, and WEST.
   - It's a simple enum without any additional methods or properties.
   - It's likely used to keep track of the current facing direction of each robot.

2. `MovementInstruction.java`:
   - This enum represents the three possible movement instructions: TURN_LEFT, TURN_RIGHT, and MOVE_FORWARD.
   - Each enum constant is associated with a String value ("L", "R", and "F", respectively), which can be accessed using the `getValue()` method.
   - The String values are likely used to parse the movement instructions from an input source (e.g., a file or user input).
   - The constructor takes the String value as a parameter and assigns it to the `value` field of each enum constant.

These enums seem to be the foundation for representing the state and movement of the robots in your application. The `FacingDirection` enum keeps track of the robot's orientation, while the `MovementInstruction` enum defines the possible actions a robot can take.

handleExceptions

Here's an overview of these files:

1. `IllegalFaceDirection.java`:
   - This class extends the `RuntimeException` class and is used to represent an exception that occurs when an invalid facing direction is encountered.
   - The constructor takes a `FacingDirection` enum value as a parameter and sets the exception message to indicate that the provided direction is not valid.
   - This exception is likely thrown when the application tries to set or update the robot's facing direction with an invalid value.

2. `ValidationCommand.java`:
   - This class contains a static utility method `isValidCommand(String commands)` that checks if a given string represents a valid sequence of movement instructions.
   - The method first checks if the input string is null, returning false if it is.
   - It then constructs a regular expression pattern by iterating over the `MovementInstruction` enum values and appending their corresponding string values to the pattern.
   - The pattern is then used to match against the input string using the `matches()` method.
   - If the input string matches the pattern, the method returns true, indicating that the string contains only valid movement instructions.

3. `IllegalCommand.java`:
   - This class extends the `RuntimeException` class and is used to represent an exception that occurs when an invalid movement instruction is encountered.
   - The constructor takes a string parameter representing the invalid command and sets the exception message accordingly.
   - This exception is likely thrown when the application encounters an invalid movement instruction in the input sequence.

Overall, these exception classes and the `ValidationCommand` utility class seem to be part of the error handling and input validation mechanism in your application. The `IllegalFaceDirection` exception is used to handle invalid facing directions, while the `IllegalCommand` exception is used to handle invalid movement instructions.

The `ValidationCommand` class provides a utility method to validate the input sequence of movement instructions against the valid set of instructions defined by the `MovementInstruction` enum. This helps ensure that the application only processes valid input data.

Exception handling and input validation are essential aspects of robust software development, and your implementation demonstrates a good approach to handling and communicating errors related to invalid input and state in your Robot Race application.

Utils

1. `CalculateCorrectDirectionUtility.java`:
   - This class contains utility methods for calculating the correct facing direction based on the current direction and a movement instruction (turn left or turn right).
   - The `changeFacingDirection` method takes the current facing direction and movement instruction as input and returns the new facing direction after the turn.
   - It first validates the input using the `validateInput` method, which checks for null values and ensures that the movement instruction is either "turn left" or "turn right".
   - Depending on the movement instruction, it calls either the `rotateLeft` or `rotateRight` method to calculate the new facing direction.
   - The `rotateLeft` and `rotateRight` methods implement the logic for rotating the direction 90 degrees left or right, respectively.
   - These methods handle all possible cases for the `FacingDirection` enum and throw an `IllegalArgumentException` if an unsupported direction value is encountered.

2. `ModelUtility.java`:
   - This class contains a utility method `convertStringToCharList` that converts a given string to a list of characters.
   - The method creates an anonymous implementation of the `AbstractList` class.
   - The `get` method of the anonymous class returns the character at the specified index of the input string.
   - The `size` method returns the length of the input string.
   - This utility method is likely used to convert the movement instructions from a string to a list of characters for easier processing.

The `CalculateCorrectDirectionUtility` class seems to be the core component responsible for handling the robot's facing direction changes based on the movement instructions. It encapsulates the logic for rotating the direction left or right and performs input validation to ensure the correctness of the operations.

The `ModelUtility` class provides a helper method for converting a string to a list of characters, which might be useful for processing the movement instructions or other string-related operations in the application.

Overall, these utility classes follow good practices by separating concerns and encapsulating reusable logic in separate classes. The input validation checks in `CalculateCorrectDirectionUtility` also contribute to the robustness of the application by handling edge cases and invalid inputs.

Models

Let's go through each file:

1. `Coordinate.java`:
   - This class represents a 2D coordinate point with `x` and `y` values.
   - It provides constructors to create a coordinate with default (0, 0) or specific (x, y) values.
   - Getter and setter methods are available to access and modify the coordinate values.
   - The `toString()` method is overridden to provide a string representation of the coordinate.

2. `Robot.java`:
   - This class implements the `RobotInterface` and represents a robot in the race.
   - It has two fields: `coordinate` (a `Coordinate` object) and `currentFacingDirection` (an enum of type `FacingDirection`).
   - The constructor takes an initial facing direction as a parameter and initializes the robot's coordinate to (0, 0).
   - The `move(String instructions)` method processes a sequence of movement instructions.
     - It performs input validation and throws exceptions for null, empty, or invalid instruction strings.
     - It iterates through each character in the instruction string and calls the corresponding movement method (`moveForward()`, `turnLeftOperation()`, or `turnRightOperation()`).
   - The `turnRightOperation()` and `turnLeftOperation()` methods update the `currentFacingDirection` based on the current direction and the turn instruction.
   - The `moveForward()` method updates the robot's coordinate based on the current facing direction.
   - Getter methods are provided to retrieve the robot's current coordinate and facing direction.

3. `RobotInterface.java`:
   - This is an interface that defines the contract for a robot in the race.
   - It declares three methods: `move(String instructions)`, `getCoordinate()`, and `getCurrentFacingDirection()`.

4. `RobotRace.java`:
   - This class extends the `Robot` class and implements the `Runnable` interface, allowing each robot to run in its own thread.
   - It has additional fields like `name`, `rank`, `delay`, and `instructions`.
   - The constructor takes the initial facing direction, robot name, delay time, and movement instructions as parameters.
   - The `run()` method is the entry point for the robot's thread.

- It performs input validation and throws exceptions for null, empty, or invalid instruction strings.
- It iterates through each character in the instruction string and calls the corresponding movement method (`moveForward()`, `turnLeftOperation()`, or `turnRightOperation()`).
- It introduces a delay between each movement based on the robot's configured delay time.
- It updates the robot's rank and coordinates when the race ends.
- It prints the final rankings and coordinates of all robots.
- The `getElapsedTime()` method returns the elapsed time since the start of the race.
- The `rankHolder` is a static queue that stores the ranked robots.

The models in your application seem to be well-structured and follow object-oriented principles. The `Robot` class encapsulates the robot's state and behavior, while the `RobotRace` class extends it and adds the threading and ranking functionality. The use of an interface (`RobotInterface`) promotes code reusability and modularity.

The input validation and exception handling mechanisms are implemented to handle invalid input and edge cases. Overall, the code appears to be well-organized and follows good programming practices.