University of Miskolc Full-time programme
Department of Information Technology

GEIAL31A-B2a
Java programming

1. lab: class, enum

1.
Create a 2D Point class in the example1 package, which:
- can be initialized with two coordinates (real numbers)
- can be initialized without parameters (coordinates); in this case both coordinates are 0,0
- can return the point's data as a String in the form "(x, y)"
- can return the distance from the origin
- can return the distance between two points given as parameters

Create an executable class PointRun in the PointRun package, in which the main method:
- creates two Point objects with random coordinates
- prints their data
- prints the data of the point that is farther from the origin
- prints the distance between the two points

Create an executable class PointsRun in the PointsRun package, in which the main method:
- creates 10 Point objects with randomly generated coordinates
- prints the points' coordinates and their distance from the origin
- prints the coordinates of the point that is farthest from the origin

2.
Create a class called Temperature, in which:
- store a temperature value in Kelvin/Celsius
- define a enum type unit with the members CELSIUS and KELVIN
- override the toString() that it returns the temperature value and the texts "C" and "K"
- can be initialized by a temperature value and a unit of measure
- can be initialized by only a temperature value, in which the unit is automatically initialized Kelvin unit
- can be determined whether a Temperature object (parameter) has a higher temperature or not
- a new Temperature object can be obtained that contains the sum of two Temperature objects (parameters)
- the temperature value can be returned in the C/K unit (specified as a parameter)
- temperature object (parameter) can be returned expressed in a unit specified as a parameter

Create an executable class called TemperatureRun, in which:
- create 10 Temeprature objects with randomly generated Celsius temperature values between 0 and 100
- prints them in Kelvin
- then prints the data of the largest (highest) temperature object
- then prints the average temperature

2. lab: arrays, reading from console

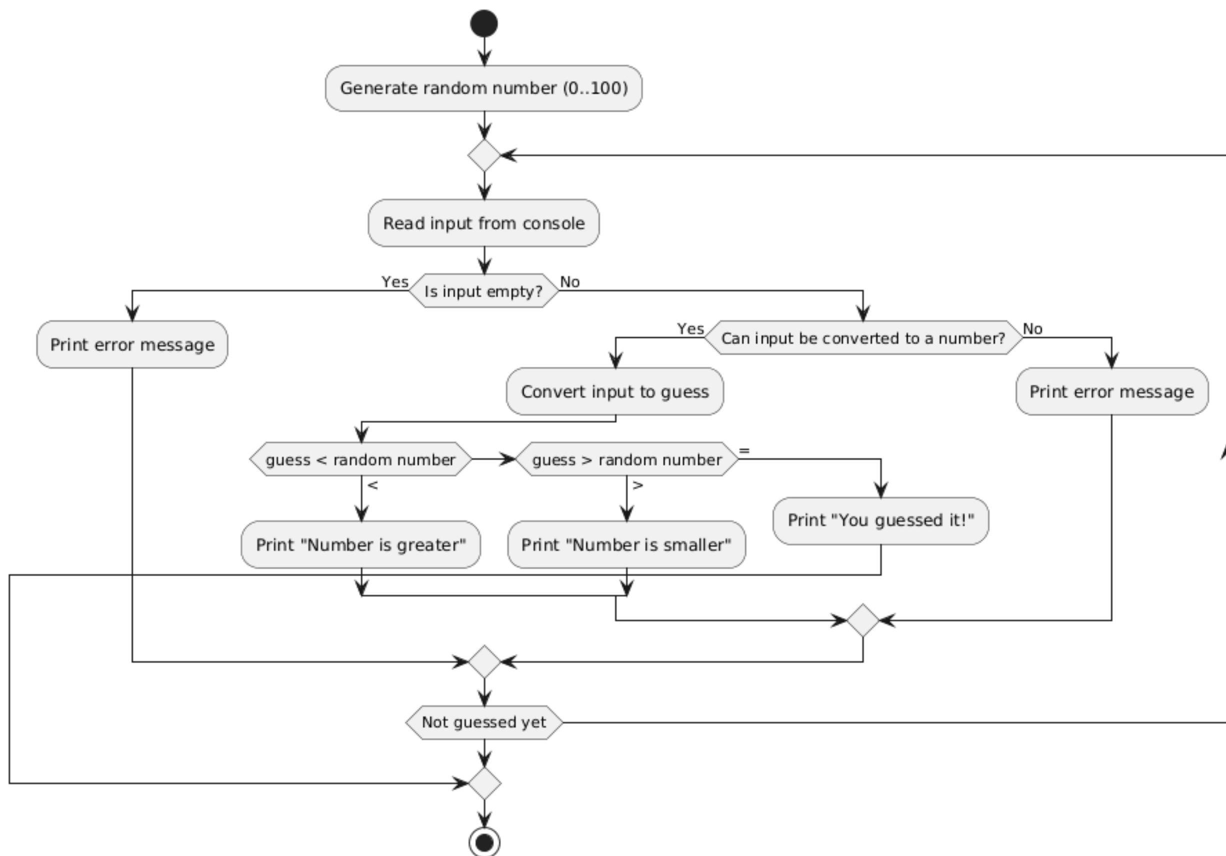
1.
Create a complete console application that stores the names and salaries of 4 people in two separate arrays (one array for names, one array for salaries).

Tasks:
- Print the data sorted by name
- Print the data sorted by salary
- Write a search algorithm to check whether a person with a given full name exists
- Write a search algorithm to print the names of people who have a given first name

Program logic
1. Create a 4-element String array (name it name)
2. Create a 4-element int array (name it salary)
3. Fill both arrays with data
4. Sort the contents of the arrays by name (method name: sortName)
5. Print the data.
6. Sort the contents of the arrays by salary (method name: sortSalary)
7. Print the data.
    - create a method called print for printing the arrays
8. Implement a search method by full name (method name: searchByFullName) that also prints the result
9. Implement a search method by first name (method name: searchByFirstName) that also prints the result

2. Number guessing game
Create a complete console application that generates a random number between 0 and 100 (both bounds inclusive).
The task is to guess the number.
The user enters a guess, and the program displays one of the following messages:
- The number is smaller than your guess!
- The number is greater than your guess!
- You guessed it!

Algorithm description
1. Generate a random number (rndNum).
2. Read a number from the console.
3. Examine the entered input:
    - Was any input provided?
    - Can the input be converted to a number?
4. If the input can be converted to a number, convert it and store it as guess.
5. If no input was provided or it cannot be converted to a number, display an error message and return to step 2.
6. Compare guess with rndNum and display the evaluation.
7. If the number was not guessed correctly, return to step 2.
8. If the number was guessed correctly, terminate the program.



3. lab: collections, comparator

1.

Book class, which stores the data of a book: data members: author, title, price

- Constructor: initialization by providing all three data members
- Override the toString(): return the book's data in the following format: “author, title, price”
- Override equals() (equality): two books should be considered equal if the author and the title match
- Implement Comparable: the natural ordering should be by author in ascending order; if the author is the same, then by title in ascending order
- Nested Comparator: the Book class should have a public static nested class, e.g. PriceComparator, which implements Comparator and compares based on price

Bookshelf class, which stores any number of books

- Add new book: add a new Book to the shelf
- Number of stored books: query how many books are on the shelf
- Retrieve book by author and title: get the corresponding book by providing a specific author + title (if not found, should indicate somehow, e.g. null)
- Retrieve array of books: get the array of books (Book[])
- Delete by Book object: delete a specified Book object from the shelf
- Delete by author and title: delete a book from the shelf by providing a specific author + title
- Retrieve sorted array (by author/title): get the array of books sorted by author/title (based on Comparable)
- Get the list converted to an array ordered by price using the Java8 compatator
- Most expensive book: get the most expensive book on the shelf

Create a runnable class (e.g. BookshelfRun), in which you test the functionality of Bookshelf

4. lab: generic type, abstract class, interface

1.
Create an abstract class Shape in the shapes package, which:

- stores the name of the shape
- create the constructor to set the data members
- has an abstract method to calculate the area
- has a concrete method that returns the name of the shape
- override the toString() method

Create two concrete subclasses of Shape:
Circle

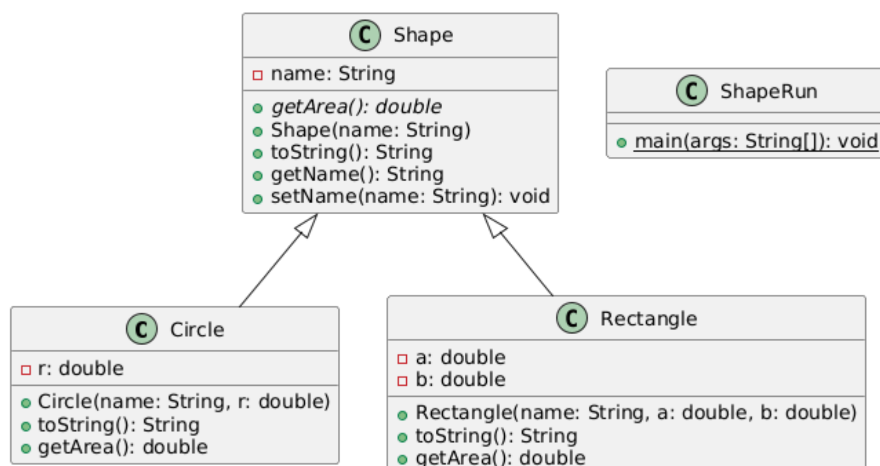
- can be initialized with a name and a radius
- overrides the area calculation method

Rectangle

- can be initialized with a name, width ("a"), and height ("b")
- override the area calculation method

Create an executable class ShapeRun in the shapesrun package, in which the main method:

- creates several Shape objects (Circle and Rectangle)
- stores them using in the same Shape reference
- prints the name and area of each shape
- demonstrates polymorphic behavior using the abstract base class



2.
Create an interface Payable in the payable package, which:

- declares a method that returns a payable amount

Create two classes that implement the Payable interface:
Employee

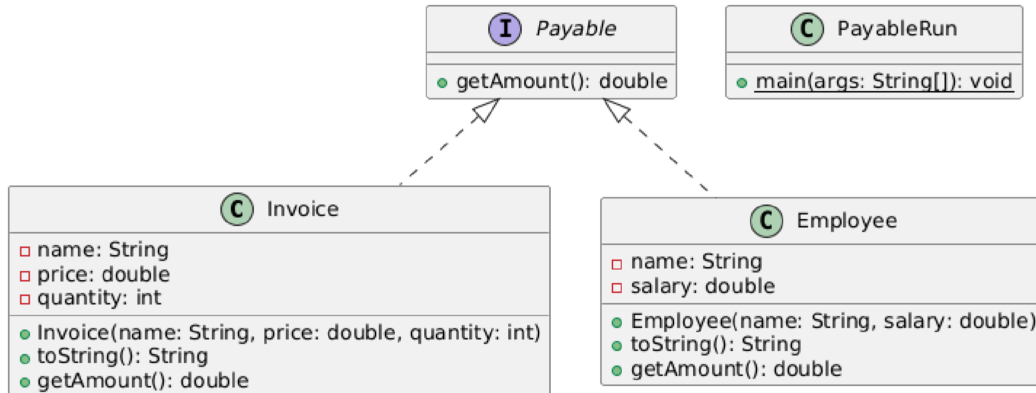
- can be initialized with a name and a salary (create fields and constructor)
- returns the salary as the payable amount

Invoice

- can be initialized with a product name, quantity, and unit price (create fields and constructor)
- returns the total price (quantity × unit price) as the payable amount

Create an executable class PayableRun in the payablerun package, in which the main method:

- creates several Payable objects (Employee and Invoice)
- stores them using Payable references
- prints the payable amount of each object
- calculates and prints the total payable amount



3.
Create a generic class called Box<T>, which can store an element of any type.

The class Box<T> must:

- store a value of generic type T
- have a method to set the stored value
- have a method to get the stored value
- override the toString() method

Create an executable class called BoxRun, in which the main method:

- creates a Box<Integer> and stores an integer value
- prints the stored integer value
- creates a Box<String> and stores a string value
- prints the stored string value
- creates a Box<Double> and stores a double value
- prints the stored double value

5. lab: inheritance, binding, exception

1. is-a relationship
Create a base class Person with:

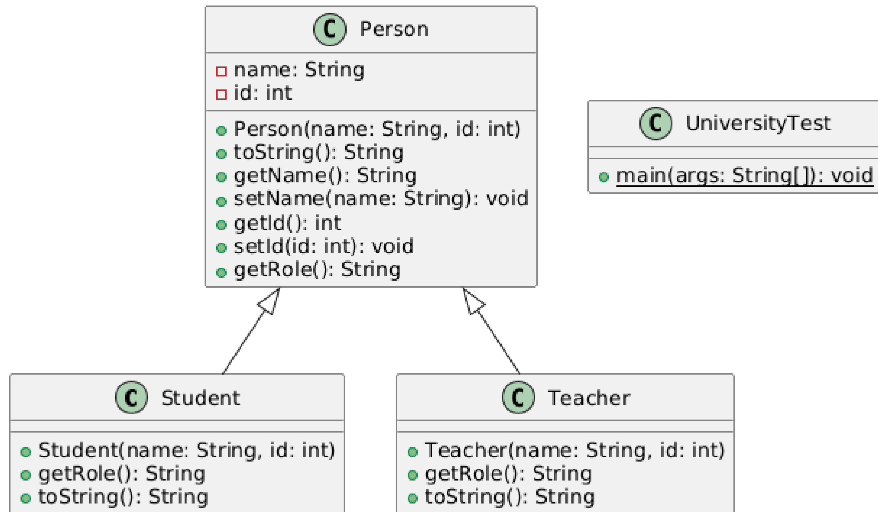
- fields: name, id
- constructor, toString, getter, setter
- method: getRole() returning "Person"

Create subclasses:

- Student: constructor, toString, overrides getRole() to return "Student"
- Teacher: constructor, toString, overrides getRole() to return "Teacher"

Create a runnable class UniversityTest:

- store Student and Teacher objects in an ArrayList<Person>
- iterate through the collection and print getRole()



2. has-a relationship

Create class CPU

- field: clock (int)
- constructor, toString, getter, setter
- method process(): write “CPU is working”

Create class Memory

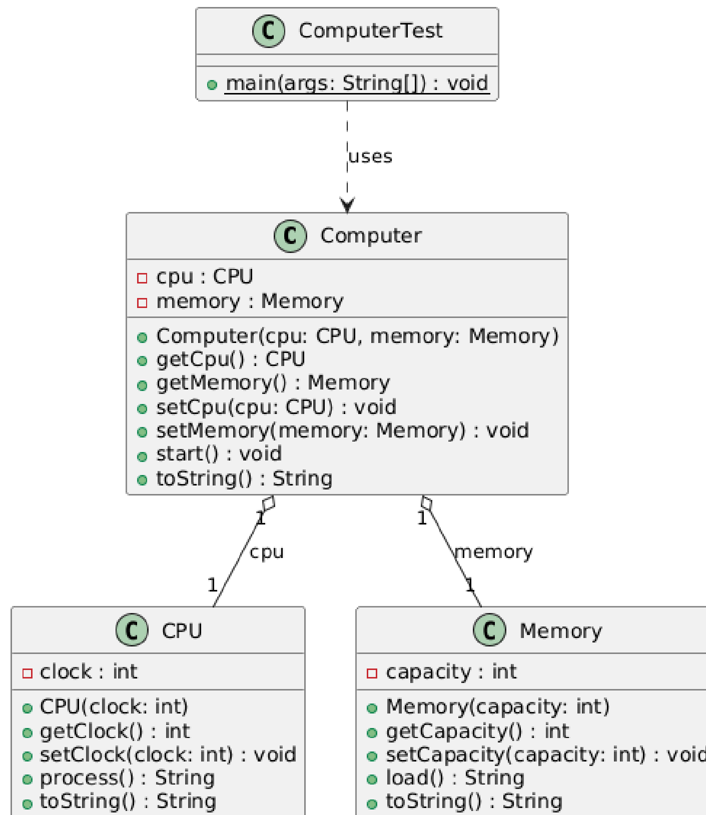
- field: capacity (int)
- constructor, toString, getter, setter
- method load(): write “Memory is under load.”

Create class Computer that:

- has a CPU and a Memory fields
- constructor, toString, getter, setter
- method start() calls both cpu.process() and memory.load()

Create a runnable class ComputerTest

- create CPU and Memory objects
- create a Computer object
- print all objects data



3. Exceptions

Create a user registration system that validates input data.

Create class User

- fields: username (String), age (int)
- constructor: initializes all fields

Create the following custom exceptions:

1. InvalidAgeException
    - thrown if age < 18
    - extend Exception (checked)

2. InvalidUsernameException
    - thrown if username is null or shorter than 5 characters
    - extend RuntimeException (unchecked)

Create class UserValidator

Validation rules:

- if username is invalid: throw InvalidUsernameException
- if age < 18: throw InvalidAgeException

Create a runnable class RegistrationTest

- create multiple User objects:
    - valid user
    - user with invalid age
    - user with invalid username
- Use try-catch with multiple catch blocks
- Print which exception was caught

4. Complex assignment
Create a small Vehicle Rental System.

Create base class Vehicle

- fields: id (int), brand (String), rented (boolean)
- constructor: initializes all fields
- public void drive(): print the "Vehicle is moving" message
- public boolean isRented()
- public void setRented(boolean rented)
- public void rent()
    - if rented == true, throws VehicleAlreadyRentedException
    - otherwise sets rented to true (mark vehicle as rented)
- public Vehicle returnVehicle()
    - returns the current Vehicle object
- public void printCategory(): “This is a vehicle.”

Create class Car

- extends Vehicle
- fields: engine (Engine)
- constructor
- overrides drive() to print: "Car is driving on the road"
- overrides printCategory(): “This is a car.”
- public void setEngine(Engine engine)
- public Engine getEngine()

Create class Engine

- fields: volume : int
- Constructor
- public void start()
- Getter, setter, and toString()

Create class Bike

- extends Vehicle
- fields: type (String)
- constructor
- overrides printCategory(): “This is a Bike.”
- overrides drive() to print: "Bike is being ridden"
- getter, setter, toString

Create custom exception: VehicleAlreadyRentedException

- extends Exception (checked exception)
- stores an error message
- thrown by Vehicle.rent() if the vehicle is already rented

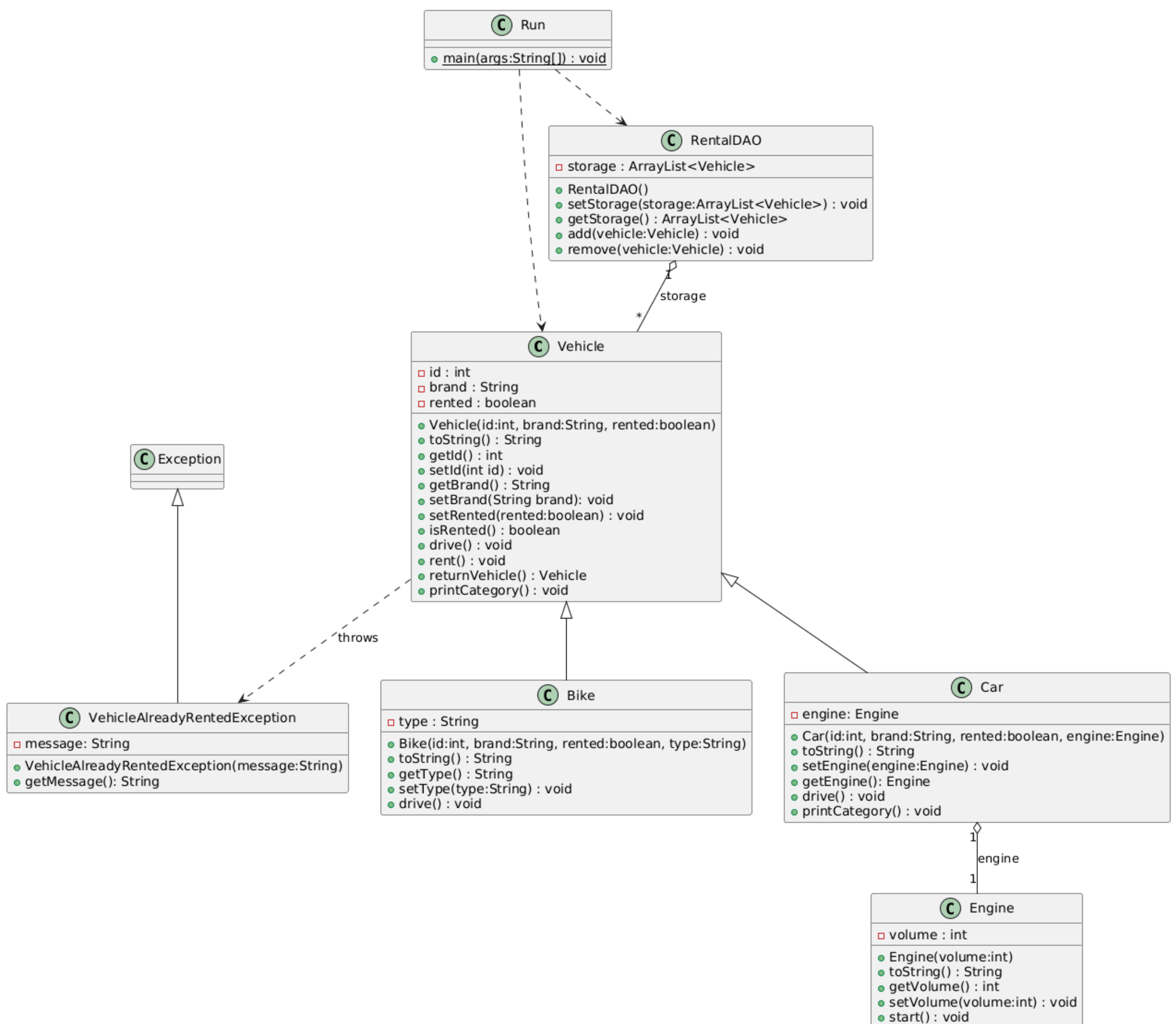
Create class RentalDAO (this class is responsible for storing and managing vehicles).
The relationship between RentalDAO and Vehicle is aggregation (the DAO stores vehicles).

- fields: storage : ArrayList<Vehicle>
- public void setStorage(ArrayList<Vehicle> storage)
- public ArrayList<Vehicle> getStorage()

- public void add(Vehicle vehicle)
- public void remove(Vehicle vehicle)

Create a runnable class RentalTest with main method:

- create an ArrayList<Vehicle>
- add at least:
    - one Car
    - one Bike
    - one Vehicle
- iterate through the list:
    - call drive() on each object
- try to:
    - rent the same vehicle twice
    - catch the custom exception
- print meaningful error messages



Remarks:

- Vehicle: Base class representing a rentable vehicle with common attributes and operations (id, brand, rental state, driving and renting behavior)
- Car: A specific type of Vehicle that has an Engine and overrides vehicle behavior (e.g. drive(), printCategory())
- Bike: A specific type of Vehicle with an additional type attribute and its own drive() implementation
- Vehicle → Car, Bike: Inheritance (is-a relationship); method overriding enables dynamic (runtime) binding
- Engine: Represents the technical engine component used by a Car.
- Car o-- Engine: Aggregation (has-a relationship); the Engine can exist independently and can be replaced
- VehicleAlreadyRentedException: Custom checked exception indicating an invalid rental attempt
- Vehicle ..> VehicleAlreadyRentedException: Dependency; the Vehicle.rent() method may throw this exception
- RentalDAO: Data access object responsible for storing and managing vehicles.
- RentalDAO o-- Vehicle: Aggregation; the DAO stores multiple Vehicle objects in a collection

6. lab: file handling

1.
Create a program that makes a copy of a file whose name is provided, with the following rules:

a) Filename modification
   insert the token MOD before the file extension
   - example: apple.txt → appleMOD.txt

b) Content transformation
   - in the copied file, every word that starts with a lowercase letter must be changed so that it starts with an uppercase letter (e.g., apple → Apple, hello → Hello)
   - words that already start with an uppercase letter should remain unchanged
   - preserve the rest of the word as is (only change the first letter)

c) General requirements
   - handle file errors gracefully (e.g., “file not found”)
   - use try-with-resources for file I/O
   - keep the original file unchanged; apply all modifications only in the new *MOD.txt file

2.
Create a program that reads Person objects from a text file.

- the file contains lines in the following format:
  - Name|date (example: Michael Poulsen|1988.10.13.)

- read all persons from the specified file
- store the loaded persons in a People (or Persons) container object
- print all persons ordered by date of birth
- also print all persons whose birthday will occur within the next 30 days