Instructions

This exercise has to be completed and submit it the day of the next lecture (deadline is 14-02-2024 23:59). To deliver the exercises, upload only the `.java` files containing your solution in Autolab.

It is possible to work in pairs but, in this case, each person has to hand in an individual submission. Additionally, during the next session, you might be asked to explain your solution. In case you are not able to properly explain the solution and answer related questions, the whole exercise will be considered as failed.

Exercise 1 - Car Collision

Implement the hierarchy of classes and corresponding logic needed to have the "Car Collision Game" program (see below) properly functioning. You are not allowed to alter in any way the provided main method.

The idea of the game is to resemble Mario Kart, where a car runs, and it can hit a truck, a pillar, or a life. If the car hits a truck or a pillar the number of lives of the car is decreased (according to the intensity). If the car hits a life, then the number of lives is incremented by a certain amount (according to the intensity). The game continues as long as the car has lives.

```
import java.util.Random;


// ...
// write here all missing classes
// ...


public class CarCollisionGame {
  public static void main(String[] args) {

    Random random = new Random();
```

```
  if (args.length > 0) {
    random.setSeed(Long.parseLong(args[0]));
  }
  Car c = new Car();
  c.setLives(10);

  Score s = new Score();
  while(c.hasLives()) {
    if (random.nextDouble() < .75) {
      System.out.println("Ouch! Obstacle hit!");
      Obstacle o = null;
      double r = random.nextDouble();
      if (r < 0.4) {
        o = new Truck(); // this should decrease the number of lives
        System.out.println("  That was a truck!");
      } else if ( r > 0.6) {
        o = new Pillar(); // this should decrease the number of lives
        System.out.println("  That was a pillar!");
      } else {
        o = new Life(); // this should increase the number of lives
        System.out.println("  That was a new life! Hurray :)");
      }
      o.setIntensity(1 + random.nextInt(3));
      c.hit(o);
      System.out.println("  Car has now " + c.getLives() + " lives");
    } else {
      System.out.println("No obstacles hit");
    }
    s.increment();
  }

  System.out.println("Game over");
  System.out.println("Final score is " + s);
}
```

In the very last line, `s` is printed after being concatenated to the string `"Final score is "`. This behavior causes the `toString()` method to be implicitly called on the `s` object (so, `s.toString()`). You can read more about this on the official Java documentation. *Tip:* as other methods, `toString()` can be overridden.

Exercise 2 - Chess

Implement the hierarchy of classes and corresponding logic needed to have the "Chess" program listed below properly functioning. The program is properly functioning when it prints a long sequence of `Test passed!`.

You are not allowed to alter in any way the provided main method. The program must properly determine the validity of possible positions of pieces on a Chess board. We assume no other pieces are on the board. The moves to check correspond to moves of king, rook, bishop, knight, and pawn. For further information you can visit the Chess entry on Wikipedia.

```
class Piece {

  // ...

  /**
   * This method does not have to check the validity of the position
   */
  public void setArbitraryPosition(Position currentPosition) {
    // ...
  }

  /**
   * This method checks if the position is a valid position
   */
  public boolean isValidPosition(Position newPosition) {
    // ...
  }
}
```

```
// ...
// write here all missing classes
// ...


public class Chess {

  public static void main(String[] args) {
    Player p1 = new Player("White player");
    p1.setColorWhite(true);
    Player p2 = new Player("Black player");
    p2.setColorWhite(false);

    System.out.println("Testing kings:");
    Piece whiteKing = new King(p1);
    whiteKing.setArbitraryPosition(new Position('f', 5));
    test(whiteKing, 'a', 1, false);
    test(whiteKing, 'f', 4, true);

    System.out.println("Testing rooks:");
    Rook blackRook = new Rook(p2);
    blackRook.setArbitraryPosition('d', 5);
    test(blackRook, 'h', 5, true);
    test(blackRook, 'h', 1, false);
    test(blackRook, 'd', 9, false);

    System.out.println("Testing bishops:");
    Piece whiteBishop = new Bishop(p1);
    whiteBishop.setArbitraryPosition(new Position('d', 5));
    test(whiteBishop, 'b', 2, false);
    test(whiteBishop, 'a', 8, true);

    System.out.println("Testing knigts:");
    Knight blackKnight = new Knight(p2);
    blackKnight.setArbitraryPosition('d', 4);
    test(blackKnight, 'e', 6, true);
```

```
    test(blackKnight, 'f', 6, false);
    test(blackKnight, 'c', 2, true);
    test(blackKnight, 'i', 8, false);

    System.out.println("Testing pawns:");
    Pawn whitePawn = new Pawn(p1);
    Pawn blackPawn = new Pawn(p2);
    blackPawn.setArbitraryPosition('b', 4);
    test(blackPawn, 'b', 3, true);
    test(blackPawn, 'b', 5, false);
    whitePawn.setArbitraryPosition('f', 2);
    test(whitePawn, 'f', 3, true);
    test(whitePawn, 'f', 4, true);
    blackPawn.setArbitraryPosition('g', 5);
    test(blackPawn, 'g', 4, true);
    test(blackPawn, 'g', 3, false);
    whitePawn.setArbitraryPosition('e', 7);
    test(whitePawn, 'd', 8, false);
    test(whitePawn, 'f', 8, false);
  }

  public static void test(Piece p, char x, int y, boolean valid) {
    if (p.isValidPosition(new Position(x, y)) == valid) {
      System.out.println("  > Test passed!");
    } else {
      System.out.println("  X Test NOT passed!");
    }
  }
}
```