Algorithmic Domain Translation Series
Volume 1 - Airport Boarding Simulation

Basic Aisle Dynamics - Publication Edition - Boolean Identity Version

Purpose of this publication

This guide translates the original aisle-shifting algorithm into a real-world airport boarding simulation while deliberately preserving the boolean identity structure of the source code. The large decorative aircraft overview has been removed so the guide can focus on the algorithm, the aircraft-door boundary, validation scenarios, and the reference implementation.

1. Series Template: Algorithm -> Domain Translation -> Same Algorithm

The aim of the Algorithmic Domain Translation Series is to show that the same computational structure can appear in different real-world systems. The code is not merely renamed; it is reinterpreted while preserving movement logic, phase resets, finish detection and identity-tracking behaviour.

Series component	Purpose
Real-world process introduction	Explain the domain before the code is introduced.
Scope boundary	State exactly where the simulation starts and ends.
Variable translation	Show how original names map to domain names.
Boolean interpretation	Explain why each boolean still exists in the domain version.
Validation scenarios	Show what each input pattern tests.
Annotated code	Keep original fragments inline beside the domain equivalent.

Why the boolean version is intentionally preserved

A cleaner object-oriented model could represent each passenger directly. This publication intentionally avoids that because the learning value is in seeing how anonymous 1s can still be treated as continuing passengers across repeated scan phases.

2. Airport Process and Simulation Boundary

A real passenger journey contains many stages before the aircraft aisle is reached. The algorithm does not simulate every airport process; it begins only when the passenger crosses the aircraft door and becomes part of the internal aircraft aisle flow.

Stage	Included in code?	Reason
Airport entrance, check-in, security, gate	Context only	These stages can later generate the starting dataset, but they are not simulated by the aisle engine.
Aircraft door entry	Simulation start	The passenger becomes a 1 in the aircraft aisle array.
Aircraft aisle movement	Fully simulated	Passengers move if the next tile is empty.
Blocking and stop-start movement	Fully simulated	Congestion emerges from repeated deterministic scans.
Assigned row / final seating zone	Simulation end	The passenger leaves the aisle model and is counted as finished.

Aircraft-door boundary

Everything before the aircraft door is context or dataset preparation. Everything after the aircraft door is simulation. Therefore, queue formation, congestion, passenger continuation, finish detection and total moves refer to the aircraft aisle after entry through the aircraft door.

3. Core One-to-One Translation

Original code concept	Airport boarding equivalent	Why it matters
people[]	boardingAisle[]	The binary aisle remains the main simulation state.

1	anonymous passenger	Passengers remain anonymous, so identity must be inferred.
0	empty aisle tile	Movement is allowed only into an empty tile.
scan row	boarding tick / phase	The algorithm advances in repeated deterministic time steps.
hasPersonMove	hasPassengerMove	Records that a passenger has started moving in this phase.
hasPersonPrevMove	hasPassengerPrevMove	Tracks continuation of the same anonymous mover.
hasPersonFinishedMoving	hasPassengerFinishedBoarding	Marks that the current movement chain has ended.
recordPersonMoved[]	recordPassengerFinishedAt[]	Registers positions already accounted for during movement tracking.

4. Phenomena Covered Inside the Aircraft Door

Phenomenon inside aircraft	How it appears in the code
Aisle queue formation	Multiple 1s occupy separated or adjacent tiles and progress only when a 0 exists ahead.
Stop-and-go movement	A passenger moves during one boarding tick, then may stop when the next tile is occupied.
Passenger continuation	hasPassengerPrevMove and recordPassengerFinishedAt help infer whether a moving 1 is the same passenger across scans.
Blocking behind slow passengers	If the next aisle tile is occupied, the passenger cannot move and the chain behind may also stop.
Boundary finish	End-of-scan logic registers a passenger when the model boundary is reached.
Total boarding movement	moves counts the total single-tile movements required by the current scenario.
Average move per passenger	moves divided by numPassengersMoving provides a simple movement efficiency metric.
Direction comparison	The program compares towardRear and towardFront outcomes.
Compact queue estimate	The analytical report evaluates how many moves are needed to make all passengers adjacent anywhere in the aisle.

5. Validation Scenarios

Scenario	Test case	What it validates
A - Active mixed aisle	int[] boardingAisle = {0,1,1,0,1,0,0,0,1};	Several anonymous passengers already inside the aircraft aisle. Tests movement identity and direction comparison.
B - Alternating passengers	int[] boardingAisle = {1,0,1,0,1};	Tests repeated one-tile opportunities and stop-start waves.
C - Sparse aisle	int[] boardingAisle = {0,0,1,0,0,0,1,0};	Tests separated movers without inventing unnecessary interactions.
D - Fully blocked aisle	int[] boardingAisle = {1,1,1,1,1};	Tests the no-movement condition.
E - Chain continuation	int[] boardingAisle = {0,1,0,0,1,0,1};	Tests whether a moving anonymous passenger is a new mover or a continuation.
F - Single passenger	int[] boardingAisle = {0,1,0};	Tests the simplest identity case.
G - Empty aisle	int[] boardingAisle = {0,0};	Confirms that the code avoids false movement.

6. Worked Aircraft-Aisle Movement Example

Initial inside-aircraft aisle state: 0 1 0 1 0 1 0

Boarding tick 1: passengers move into open tiles ahead where possible -> 0 0 1 0 1 0 1

Boarding tick 2: remaining gaps close toward the target direction -> 0 0 0 1 0 1 1

Boarding tick 3: movement continues until the finish condition is met -> 0 0 0 0 1 1 1

Interpretation

The code does not simply count 1s. It also interprets movement over time: a 1 that moves in one tick may need to be treated as the same anonymous passenger in the next tick.

7. Reference Implementation

```
/*
================================================================================
AIRPORT BOARDING SIMULATION - BOOLEAN IDENTITY VERSION
================================================================================
Purpose of this version:
- Keep Amit's original aisle-shifting structure very close.
- Convert the story to airport boarding.
- Preserve the important boolean-style identity logic inline.
- Keep original code fragments commented beside the airport equivalent where possible.

Core mapping:
   people[]              -> boardingAisle[]
   1                     -> anonymous passenger in aisle
   0                     -> empty aisle tile
   person                -> passenger
   move right              -> board toward rear seats
   move left             -> deplane / move toward front door
   scan row              -> boarding tick / boarding phase
   finished moving         -> reached final packed boarding zone
   recordPersonMoved[]     -> recordPassengerFinishedAt[]

Important limitation:
This version intentionally keeps passengers anonymous as 1s.
That is why the boolean logic remains important: identity must be inferred through scans.
================================================================================
*/

import java.util.Arrays;

class Main
{
   public static void main(String[] args)
   {
      AirportBoardingBooleanIdentity sim = new AirportBoardingBooleanIdentity();
      sim.boardLeftOrRightVerdict();
      sim.reportMinimumAdjacentBoardingZoneMoves();
   }
}

class AirportBoardingBooleanIdentity
{
   // ============================================================
   // TEST CASES AT TOP - airport boarding equivalents
   // ============================================================
   // 0 = empty aisle tile
   // 1 = anonymous passenger standing in aisle
   // The active test case below is intentionally the same shape as your original.

   int[] boardingAisle = new int[]{0, 1, 1, 0, 1, 0, 0, 0, 1};

   // Original equivalent:
   // int[] people = new int[]{0, 1, 1, 0, 1,0,0,0,1};

   // Additional boarding test cases:
   // int[] boardingAisle = new int[]{1, 0, 1, 0, 1};            // alternating passengers
   // int[] boardingAisle = new int[]{0, 0, 1, 0, 0, 0, 1, 0};    // sparse boarding aisle
   // int[] boardingAisle = new int[]{1, 1, 1, 1, 1};            // fully blocked aisle
   // int[] boardingAisle = new int[]{0, 1, 0, 0, 1, 0, 1};       // chain continuation case
   // int[] boardingAisle = new int[]{0, 1, 0, 0, 1, 0};         // two passengers, open aisle
   // int[] boardingAisle = new int[]{0, 0};                     // empty aisle
   // int[] boardingAisle = new int[]{0, 1, 0};                  // one passenger
```

```
int aisleLength = boardingAisle.length;
int numberPassengers = countPassengers();

// ========================================================
// ORIGINAL STATE VARIABLES, AIRPORT RENAMED
// ========================================================

boolean isStartAtFrontDoor = false;
// ORIGINAL:
// boolean isStartZero=false;
// AIRPORT MEANING:
// true  => scan from front door / left side toward rear / right side.
// false => scan from rear / right side toward front door / left side.

boolean hasMultiplePassengers = false;
// ORIGINAL:
// boolean hasMultiplePeople=false;
// AIRPORT MEANING:
// Set true if the proximity scan finds two passengers on one side.

boolean hasMinimumSinglePassenger = false;
// ORIGINAL:
// boolean hasMinimumSinglePerson=false;
// AIRPORT MEANING:
// Set true if at least one passenger exists anywhere in the aisle.

int count = 0;
int distanceFromFront = 0;
int distanceFromRear = 0;

// ORIGINAL:
// int distanceLeft=0;
// int distanceRight=0;

int passengerPositions[] = new int[2];
int singlePassengerLocation;

// ORIGINAL:
// int pos[]=new int[2];
// int singlePersonLocation;

double averageMoves;

int totalMovesTowardRear = 0;
int totalMovesTowardFront = 0;
int totalPassengersMovedTowardRear = 0;
int totalPassengersMovedTowardFront = 0;
double averageMovesTowardRear = Double.NaN;
double averageMovesTowardFront = Double.NaN;

// ORIGINAL EQUIVALENT:
// int totalMovesRight = 0;
// int totalMovesLeft  = 0;
// int totalPeopleMovedRight = 0;
// int totalPeopleMovedLeft  = 0;
// double averageMovesRight = Double.NaN;
// double averageMovesLeft  = Double.NaN;

public AirportBoardingBooleanIdentity()
{
```

```
    int farEnd;
    int start = 0;
    String boardingDirection;

    System.out.println("AIRPORT BOARDING BOOLEAN IDENTITY SIMULATION");
    System.out.println("Aisle length is: " + aisleLength);
    System.out.println("Number passengers is: " + numberPassengers);
    System.out.println("Original boarding aisle: " + Arrays.toString(boardingAisle));
    System.out.println("0 = empty aisle tile, 1 = anonymous passenger");
    System.out.println();

    System.out.println("A proximity technique from front and rear estimates a likely boarding direction.");
    System.out.println("IMPORTANT: this heuristic is not always correct, so full simulations are compared.");

    System.out.println("\n**** Calculating proximity of two passengers from FRONT DOOR side ****");
    farEnd = aisleLength;

    if (start == 0)
    {
       isStartAtFrontDoor = true;
       boardingDirection = "towardRear";
    }
    else
    {
       boardingDirection = "towardFront";
    }

    scanSideToSide(start, farEnd);
    System.out.println("********** BOARDING ALL PASSENGERS: " + boardingDirection);
    beginBoardingMove(start, farEnd, boardingDirection);

    isStartAtFrontDoor = false;

    System.out.println("\n**** Calculating proximity of two passengers from REAR side ****");
    start = aisleLength - 1;
    farEnd = 0;

    if (start == 0)
    {
       isStartAtFrontDoor = true;
       boardingDirection = "towardRear";
    }
    else
    {
       boardingDirection = "towardFront";
    }

    scanSideToSide(start, farEnd);
    System.out.println("********** BOARDING ALL PASSENGERS: " + boardingDirection);
    beginBoardingMove(start, farEnd, boardingDirection);
}

public int countPassengers()
{
    int total = 0;
    for (int i = 0; i < aisleLength; i++)
    {
       if (boardingAisle[i] == 1)
       {
          total++;
       }
    }
```

```
    return total;
}

public void scanSideToSide(int start, int otherEnd)
{
    // ORIGINAL:
    // public void sideToSide(int start, int otherEnd)

    for (int i = start;
         isStartAtFrontDoor ? i < otherEnd : i >= otherEnd;
         i = isStartAtFrontDoor ? i + 1 : i - 1)
    {
        System.out.println("aisle tile " + i + " contains: " + boardingAisle[i]);

        if (count == 2)
        {
            break;
        }

        if (boardingAisle[i] == 1)
        {
            passengerPositions[count] = i;
            count++;

            if (count == 1)
            {
                singlePassengerLocation = i;
                hasMinimumSinglePassenger = true;
            }
        }
    }

    if (count == 2)
    {
        if (start == 0)
        {
            distanceFromFront = passengerPositions[1] - passengerPositions[0] - 1;
            System.out.println("Distance from front-side pair is: " + distanceFromFront + "\n");
        }
        else
        {
            distanceFromRear = passengerPositions[0] - passengerPositions[1] - 1;
            System.out.println("Distance from rear-side pair is: " + distanceFromRear + "\n");
        }
        hasMultiplePassengers = true;
    }
    else if (count == 1)
    {
        System.out.println("One passenger in aisle: location " + singlePassengerLocation);
    }
    else
    {
        System.out.println("Empty aisle");
    }

    count = 0;
}

public void boardLeftOrRightVerdict()
{
    // ORIGINAL:
    // public void leftOrRight()
```

```
System.out.println("\nAMIT AMLANI ORIGINAL-STYLE BOARDING VERDICT");
System.out.println("Reason: only inspects the first two passengers from each side.");

if (hasMinimumSinglePassenger)
{
   if (distanceFromFront > distanceFromRear)
   {
      System.out.println("\n******VERDICT: Board passengers toward rear\n");
   }
   else if (distanceFromRear > distanceFromFront)
   {
      System.out.println("\n******VERDICT: Move passengers toward front\n");
   }
   else
   {
      if (distanceFromFront == 0 && distanceFromRear == 0 && !hasMultiplePassengers)
      {
         if ((aisleLength - 1 - singlePassengerLocation) < singlePassengerLocation)
         {
            System.out.println("\n******VERDICT: Move single passenger toward rear\n");
         }
         else if (singlePassengerLocation < (aisleLength - 1) - singlePassengerLocation)
         {
            System.out.println("\n******VERDICT: Move single passenger toward front\n");
         }
         else
         {
            System.out.println("\n******VERDICT: no difference moving single passenger\n");
         }
      }
      else
      {
         System.out.println("\n******VERDICT: no difference moving front or rear\n");
      }
   }
}

System.out.println("------------------------------------------------------------");
System.out.println("SIMULATION VERDICT => compares actual full boarding outcomes");

if (!hasMinimumSinglePassenger)
{
   return;
}

if (totalMovesTowardRear < totalMovesTowardFront)
{
   System.out.println("\n******VERDICT: Board passengers toward rear\n");
}
else if (totalMovesTowardFront < totalMovesTowardRear)
{
   System.out.println("\n******VERDICT: Move passengers toward front\n");
}
else
{
   if (!Double.isNaN(averageMovesTowardRear) && !Double.isNaN(averageMovesTowardFront))
   {
      if (averageMovesTowardRear < averageMovesTowardFront)
      {
         System.out.println("\n******VERDICT: Board passengers toward rear\n");
      }
      else if (averageMovesTowardFront < averageMovesTowardRear)
```

```
            {
               System.out.println("\n******VERDICT: Move passengers toward front\n");
            }
            else
            {
               System.out.println("\n******VERDICT: no difference moving front or rear\n");
            }
         }
         else
         {
            System.out.println("\n******VERDICT: no difference moving front or rear\n");
         }
      }
      System.out.println("-------------------------------------------------------");
   }

   public boolean checkAllPassengersFinished(boolean hasConfirmedFinishedBoarding, String boardingDirection)
   {
      // ORIGINAL:
      // public boolean checkHasFinished(boolean hasConfirmedFinishedMoving, String directionMove)

      int interestedLocation;

      for (int i = 0; i < numberPassengers; i++)
      {
         if ("towardRear".equals(boardingDirection))
         {
            interestedLocation = aisleLength - (i + 1);
         }
         else
         {
            interestedLocation = i;
         }

         if (!(boardingAisle[interestedLocation] == 1))
         {
            hasConfirmedFinishedBoarding = false;
         }
      }
      return hasConfirmedFinishedBoarding;
   }

   public void beginBoardingMove(int start, int otherEnd, String boardingDirection)
   {
      // ORIGINAL:
      // public void beginMove(int start, int otherEnd, String directionMove)

      boolean hasFinishedBoarding = true;
      // ORIGINAL: boolean hasFinishedMoving=true;

      boolean hasPassengerPrevMove = false;
      // ORIGINAL: boolean hasPersonPrevMove=false;
      // Meaning: used to identify whether this is a continuation of a passenger movement in this scan.

      boolean hasPreviousPassenger = false;
      // ORIGINAL: boolean hasPreviousPerson=false;
      // Meaning: previous anonymous passenger has just been registered as finished.

      boolean hasPassengerMove = false;
      // ORIGINAL: boolean hasPersonMove=false;
      // Meaning: a passenger has started or is currently moving during this scan.
```

```
boolean hasPassengerFinishedBoarding = true;
// ORIGINAL: boolean hasPersonFinishedMoving=true;
// Meaning: current anonymous passenger has reached a stop/finish condition.

int numPassengersMoving = 0;
// ORIGINAL: int numPeopleMoving=0;

int moves = 0;
int[] original = new int[aisleLength];
int k = 0;

System.arraycopy(boardingAisle, 0, original, 0, aisleLength);

int[] recordPassengerFinishedAt = new int[50];
// ORIGINAL: int [] recordPersonMoved = new int[50];
// Airport meaning: because passengers are anonymous 1s, this approximates identity by recording finish positions.

int numberBoardingTicks = 0;
// ORIGINAL: int numberScansRow=0;
// Airport meaning: each scan is a boarding tick / simulation phase.

int currentNumberBoardingTicks = 0;
// ORIGINAL: int currentNumberScansRow=0;

int indexMove;
int finishPoint = 0;

do
{
   numberBoardingTicks++;
   hasFinishedBoarding = true;

   // ======================================================
   // PHASE RESET - this is the key connection to your code
   // ======================================================
   // ORIGINAL FIX:
   // hasPersonPrevMove = false;
   // hasPreviousPerson = false;
   // hasPersonMove = false;
   // hasPersonFinishedMoving = true;
   //
   // AIRPORT EQUIVALENT:
   hasPassengerPrevMove = false;
   hasPreviousPassenger = false;
   hasPassengerMove = false;
   hasPassengerFinishedBoarding = true;

   System.out.println("\nBOARDING TICK " + numberBoardingTicks + " START: " + Arrays.toString(boardingAisle));

   for (int i = start;
       isStartAtFrontDoor ? i < (otherEnd - 1) : i > otherEnd;
       i = isStartAtFrontDoor ? i + 1 : i - 1)
   {
      if (boardingDirection.equals("towardFront"))
      {
         indexMove = i - 1;
      }
      else
      {
         indexMove = i + 1;
      }
```

```
if ((boardingAisle[indexMove] == 0) && (boardingAisle[i] == 1))
{
   hasPassengerMove = true;

   if (!hasPassengerPrevMove && hasPassengerMove)
   {
      if (i == (aisleLength - 1))
      {
         k = 1;
      }

      for (int m = 0; m < k; m++)
      {
         if (i == recordPassengerFinishedAt[m])
         {
            System.out.println("\n1START POS / SAME PASSENGER CONTINUES: " + Arrays.toString(boardingAisle));
            System.out.println("Same anonymous passenger continuing from aisle tile: " + i);
            hasPassengerFinishedBoarding = false;
            currentNumberBoardingTicks = numberBoardingTicks;
            break;
         }
         else if (m == (k - 1))
         {
            System.out.println("\n2START POS / NEW PASSENGER STARTS: " + Arrays.toString(boardingAisle));
            System.out.println("New anonymous passenger has commenced movement at aisle tile: " + i);
            hasPassengerFinishedBoarding = false;
            numPassengersMoving++;
            System.out.println("**************************NUMBER PASSENGERS MOVING: " + numPassengersMoving);
            currentNumberBoardingTicks = numberBoardingTicks;
         }
      }

      if (k == 0 && numberPassengers != 0)
      {
         System.out.println("\n4START POS / FIRST PASSENGER STARTS: " + Arrays.toString(boardingAisle));
         System.out.println("New anonymous passenger has commenced movement at aisle tile: " + i);
         System.out.println("**************************NUMBER PASSENGERS MOVING: " + (numPassengersMoving + 1));

         hasPassengerFinishedBoarding = false;
         numPassengersMoving++;
         currentNumberBoardingTicks = numberBoardingTicks;
      }
   }

   // ORIGINAL MOVE:
   // people[i]=0;
   // people[indexMove]=1;
   // moves++;
   // System.out.println("               " + Arrays.toString(people));
   //
   // AIRPORT MOVE:
   boardingAisle[i] = 0;
   boardingAisle[indexMove] = 1;
   moves++;
   System.out.println("               " + Arrays.toString(boardingAisle));

   hasPassengerPrevMove = true;
}
else if (hasPassengerMove && boardingAisle[i] != 0)
{
   if (numberBoardingTicks > currentNumberBoardingTicks && currentNumberBoardingTicks != 0)
   {
```

```
            currentNumberBoardingTicks = numberBoardingTicks;

            if ("towardRear".equals(boardingDirection))
            {
               finishPoint = aisleLength - 1;
            }
            else if ("towardFront".equals(boardingDirection))
            {
               finishPoint = 0;
            }

            System.out.println("PREVIOUS PASSENGER FINISHED / BLOCKED AT POSITION: " + finishPoint);
            hasPassengerFinishedBoarding = true;
            System.out.println("REGISTERING UNIQUE ID APPROXIMATION FOR PREVIOUS PASSENGER: " + finishPoint);
            recordPassengerFinishedAt[k] = 0;
            k++;
            hasPreviousPassenger = true;
         }
         else
         {
            System.out.println("PASSENGER FINISHED / BLOCKED AT POSITION: " + i);
            hasPassengerFinishedBoarding = true;
         }

         hasPassengerMove = false;
         hasPassengerPrevMove = false;

         if (numberBoardingTicks > currentNumberBoardingTicks && currentNumberBoardingTicks != 0)
         {
            System.out.println("REGISTERING UNIQUE ID APPROXIMATION FOR PASSENGER: " + 0);
         }
         else
         {
            if (!hasPassengerFinishedBoarding)
            {
               System.out.println("PASSENGER FINISHED / BLOCKED AT POSITION: " + i);
            }

            if (hasPreviousPassenger)
            {
               System.out.println("PASSENGER FINISHED / BLOCKED AT POSITION: " + i);
               hasPreviousPassenger = false;
            }

            System.out.println("REGISTERING UNIQUE ID APPROXIMATION FOR PASSENGER: " + i);
         }

         recordPassengerFinishedAt[k] = i;
         k++;
      }
   }

   // ========================================================
   // END-OF-SCAN FINISH CHECK
   // ========================================================
   // ORIGINAL:
   // if (hasPersonMove) { int endPoint = directionMove.equals("right") ? (length - 1) : 0; ... }

   if (hasPassengerMove)
   {
      int endPoint = boardingDirection.equals("towardRear") ? (aisleLength - 1) : 0;
```

```
            if (boardingAisle[endPoint] == 1)
            {
               System.out.println("PASSENGER FINISHED / REACHED BOARDING BOUNDARY AT POSITION: " + endPoint);
               System.out.println("REGISTERING UNIQUE ID APPROXIMATION FOR PASSENGER: " + endPoint);

               recordPassengerFinishedAt[k] = endPoint;
               k++;
            }

            hasPassengerMove = false;
            hasPassengerPrevMove = false;
         }

         hasFinishedBoarding = checkAllPassengersFinished(hasFinishedBoarding, boardingDirection);

      } while (!hasFinishedBoarding);

      System.out.println("Number passengers moved: " + numPassengersMoving);
      k = 0;

      System.out.println("\n\n-----------------------------------------------------------");
      System.out.println("Total number of moves " + boardingDirection + ": " + moves);
      System.out.println("FINISHED BOARDING AISLE: " + Arrays.toString(boardingAisle));
      System.out.println("Number passengers moving for average: " + numPassengersMoving);

      if (numPassengersMoving == 0)
      {
         averageMoves = 0.0;
      }
      else
      {
         averageMoves = (double)moves / (double)numPassengersMoving;
      }

      System.out.println("AVERAGE MOVE PER PASSENGER: " + averageMoves);
      System.out.println("-----------------------------------------------------------");

      if ("towardRear".equals(boardingDirection))
      {
         totalMovesTowardRear = moves;
         totalPassengersMovedTowardRear = numPassengersMoving;
         averageMovesTowardRear = averageMoves;
      }
      else
      {
         totalMovesTowardFront = moves;
         totalPassengersMovedTowardFront = numPassengersMoving;
         averageMovesTowardFront = averageMoves;
      }

      System.arraycopy(original, 0, boardingAisle, 0, aisleLength);
   }

   public void reportMinimumAdjacentBoardingZoneMoves()
   {
      // ORIGINAL:
      // reportMinimumAdjacencyMoves()
      // Airport equivalent:
      // What is the minimum number of single-tile moves to make passengers form one compact boarding queue?

      System.out.println("\n============================================================");
```

```
System.out.println("MINIMUM MOVES TO MAKE ALL PASSENGERS ADJACENT ANYWHERE");
System.out.println("============================================================");

if (numberPassengers <= 1)
{
   System.out.println("Trivial case: numberPassengers = " + numberPassengers + ". Already adjacent.");
   System.out.println("Original aisle: " + Arrays.toString(boardingAisle));
   return;
}

int[] passengerOnePositions = new int[numberPassengers];
int idx = 0;

for (int i = 0; i < aisleLength; i++)
{
   if (boardingAisle[i] == 1)
   {
      passengerOnePositions[idx] = i;
      idx++;
      if (idx == numberPassengers) break;
   }
}

System.out.println("Original aisle: " + Arrays.toString(boardingAisle));
System.out.println("Passenger positions left-to-right: " + Arrays.toString(passengerOnePositions));
System.out.println("Evaluating compact boarding blocks of size: " + numberPassengers + "\n");

int bestStart = -1;
int bestMoves = Integer.MAX_VALUE;

for (int s = 0; s <= (aisleLength - numberPassengers); s++)
{
   int total = 0;
   StringBuilder details = new StringBuilder();

   for (int j = 0; j < numberPassengers; j++)
   {
      int from = passengerOnePositions[j];
      int to = s + j;
      int d = Math.abs(from - to);
      total += d;

      if (j > 0) details.append(" | ");
      details.append("Passenger").append(j).append(": ").append(from).append("->").append(to).append(" (").append(d).append(")");
   }

   System.out.println("Block start s=" + s + " targets [" + s + "..." + (s + numberPassengers - 1) + "]"
                      + "  TOTAL MOVES=" + total);
   System.out.println("  " + details);

   if (total < bestMoves)
   {
      bestMoves = total;
      bestStart = s;
   }
}

System.out.println("\n>>> BEST COMPACT BOARDING BLOCK START = " + bestStart
                   + "  targets [" + bestStart + "..." + (bestStart + numberPassengers - 1) + "]");
System.out.println(">>> MINIMUM TOTAL MOVES = " + bestMoves);
```

```
        int[] target = new int[aisleLength];
        for (int i = bestStart; i < bestStart + numberPassengers; i++)
        {
            target[i] = 1;
        }

        System.out.println("Target arrangement: " + Arrays.toString(target));
        System.out.println("============================================================\n");
    }
}
```

Appendix A - Additional Dataset Extensions

Dataset extension	How it attaches to the engine	Boundary note
Passenger walking speed	Changes how quickly a 1 advances through open tiles.	Still begins at aircraft door.
Boarding group	Controls the order in which passengers enter the aisle.	Does not change the movement rule.
Cabin baggage volume	May create temporary stop events at a row.	External factor, not part of the core boolean identity.
Aircraft type	Changes aisle length and number of local aisle datasets.	Same engine can be reused.
Assistance passenger	May alter movement duration or require special handling.	Requires domain-specific review.

Appendix B - Research Evolution

Original Boolean Algorithm -> Volume 1 Basic Aisle Dynamics -> Volume 2 Advanced Cabin Dynamics -> Future domain translations such as hospital patient flow, warehouse logistics, manufacturing flow, emergency evacuation and other deterministic people-movement systems.

Publication Edition Statement

This publication edition freezes Volume 1 as the basic aisle-dynamics reference. Decorative aircraft overview artwork has been removed so the document remains focused on the simulation boundary, the boolean identity mapping, validation scenarios and the complete code reference.