**Q1. A.**

|  |  |
| --- | --- |
| **Test message** | **Result** |
| multiply(1,2) | 2 |
| multiply(2,0) | 0 |
| multiply(-1,3) | -3 |

**Q1. B.**

|  |  |
| --- | --- |
| **Test message** | **Result** |
| printSign(1,2) | The product is positive |
| printSign(2,0) | The product is zero |
| printSign(-1,3) | The product is negative |

**Whole class**:

/\*\*

\* Objects of the Multiplier class are integer numbers.

\* The class models them has storing two integers, labelled a and b.

\*

\* The protocol either returns the value of multiplying two integer values together.

\* Or it can print out a message relating to the value of the product of the two integers, determining if it is positive, zero or negative.

\*

\* @author (Ross Barden)

\* @version (05.12.2015)

\*/

public class Multiplier

{

private int a,b;

/\*\*

\* Constructor for objects of class Multiplier.

\*/

public Multiplier()

{

}

/\*\*

\* Returns the value of multiplying two intergers, taken as arguments.

\*/

public int multiply(int a, int b)

{

return a \* b;

}

/\*\*

\* Caculates the value of multiplying two intergers, and prints out a statement relating to the value.

\* If the product is a positive number, it prints out the message "The product is positive"

\* If the product is zero, it prints out the message "The product is zero"

\* If the product is a negative number, it prints out the message "The product is negative"

\*/

public void printSign(int a, int b)

{

int product = this.multiply(a,b);

{

if (product >= 1)

{

System.out.println("The product is positive");

}

else if (product == 0)

{

System.out.println("The product is zero");

}

else

{

System.out.println("The product is negative");

}

}

}

}

**Q2. A.**

/\*\*

\* Moves all the component parts of the train into their initial positions

\*/

private void makeTrain()

{

this.setInitialCabinPos();

this.setInitialEnginePos();

this.setInitialFunnelPos();

this.setInitialWindowPos();

this.setInitialFrontWheelPos();

this.setInitialBackWheelPos();

this.setInitialSmokePos();

}

**Q2. B.**

/\*\*

\* Moves the smoke of the receiver Train object to a home

\* position relative to the funnel.

\*/

private void setInitialSmokePos()

{

this.getSmoke().setColour(OUColour.BROWN);

this.getSmoke().setDiameter(10);

this.getSmoke().setXPos(funnel.getXPos()+5);

this.getSmoke().setYPos(funnel.getYPos()-20);

}

**Q2. C.**

/\*\*

\* Moves each component of the receiver Train object (except the smoke)

\* by numUnits to the right

\*/

public void moveTrainBy(int numUnits)

{

// to be written in Q2(c)

this.getCabin().setXPos(cabin.getXPos()+ numUnits);

this.getEngine().setXPos(engine.getXPos()+ numUnits);

this.getFunnel().setXPos(funnel.getXPos()+ numUnits);

this.getWindow().setXPos(window.getXPos()+ numUnits);

this.getFrontWheel().setXPos(frontWheel.getXPos()+ numUnits);

this.getBackWheel().setXPos(backWheel.getXPos()+ numUnits);

}

**Q2. D.**

/\*\*

\* Repeatedly moves the train numUnits to the right (except the smoke)

\* by numMoves number of times. The smoke, however moves separately as follows:

\* Every move the train makes, the smoke moves 5 units backwards and 10 units up,

\* and its diameter increases by 3. However at every sixth move, the smoke

\* is reset to its initial size and position above the funnel.

\*/

public void animateTrain(int numUnits, int numMoves)

{

// To be written in Q2(d).

// You will need to making appropriate use of a delay() message

// with an argument of about 20 to slow down the motion

int distanceRight = numUnits;

int timesRight = numMoves;

int timeDelay = 100;

int smokeReset = 6;

while (timesRight > 0)

{

this.moveTrainBy(distanceRight);

this.delay(timeDelay);

timesRight--;

if (smokeReset >0)

{

this.getSmoke().setXPos(smoke.getXPos() - 10);

this.getSmoke().setYPos(smoke.getYPos() - 5);

this.getSmoke().setDiameter(smoke.getDiameter() +3);

this.delay(timeDelay);

smokeReset--;

}

if (smokeReset == 0)

{

this.setInitialSmokePos();

smokeReset = smokeReset + 6;

}

}

}

**Q2. E.**

/\*\*

\* Gets the number of units for each move from the user, then the

\* method gets the number of moves from the user.

\* If (number of units \* number of moves) is more than the guestimated

\* width of the maximised Shapes window, the user

\* is told that the train will not run and the method ends.

\* Otherwise the train moves as required.

\*/

public void run()

{

// To be written in Q2(e)

String moveDistance = OUDialog.request("Please enter a positive number for the distance of each move");

int distance = Integer.parseInt(moveDistance);

String moveTimes = OUDialog.request("Please enter s positive number for the number of times the train will move");

int times = Integer.parseInt(moveTimes);

if (distance\*times <= 1000) // The estimated size of my screen was 1000.

{

this.animateTrain(distance, times);

}

else

{

OUDialog.alert("You have exceeded the with of the screen");

}

}

**Q2. F.**

Out of the three methods that are public the one that should remain so is Public void run(). As it moves the whole object, this means that its integrity of kept intact. The other two methods do not move the entire object (in this case the train & smoke) meaning that other classes can use them, and if they do so they could damage the integrity of the object.