**CALL REQUEST LIST**

**/\*\***

**\***

**\* @author MattCasiro**

**\* Created: 23 May 2016**

**\*/**

**public class CallRequestList**

{

private LinkedList<CallButton> requestList;

private Direction direction;

**/\*\***

**\* Instantiate a Call Request List to hold a list of waiting elevator calls**

**\* that want to move in a specified direction.**

**\* @param direction is "UP" if the list services up call buttons, otherwise "DOWN"**

**\*/**

**public CallRequestList(Direction direction) throws IllegalArgumentException**

{

if (direction == Direction.DOWN || direction == Direction.UP)

{

this.direction = direction;

} else {

throw new IllegalArgumentException();

}

requestList = new LinkedList<>();

}

**/\*\***

**\* Assign valid destinations to a given elevator.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: The elevator is assigned all valid destinations given its current state<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param e is the elevator being assigned new destinations**

**\*/**

**public void getDestinations(Elevator e)**

{

// Iterate over the destination list and add call buttons to the elevator

// if the elevator is in a position to service them

ListIterator i = requestList.listIterator();

while (i.hasNext()) {

CallButton callBtn;

callBtn = (CallButton) i.next();

// Add the floor to the elevator and remove from the request list

// if possible, otherwise catch the error to prevent a crash

try {

e.addServiceRequest(callBtn);

i.remove();

} catch (IllegalArgumentException ex) {

// crash prevention - i.remove() fails if exception is thrown

// above so list state is maintained.

}

}

}

/\*\*

\* Check if floorID should be added to an elevator's destination list.<br>

\*

\* PRE: N/A<br>

\* POST: N/A<br>

\* Cleanup: N/A<br>

\*

\* @param floorID is the integer representing the floor being checked

\* @param currentFloor is the elevator's current location

\* @return true if the elevator is going up and the floor is above it, or

\* if the elevator is going down and the floor is below it

\*/

//REMOVED: all floor requests added to initial elevator as test cases required

//immediate pick-up as elevator passes a floor with a request, regardless of

//elevator and request direction.

// private boolean isValidFloor(CallButton callBtn, Elevator e)

// {

// // Compare call button direction to elevator direction and add if matching

// if (direction == Direction.UP && e.getDirection() == Direction.UP)

// {

// return callBtn.getFloorID() > e.getCurrentFloor();

// } else if (direction == Direction.DOWN && e.getDirection() == Direction.DOWN) {

// return callBtn.getFloorID() < e.getCurrentFloor();

// } else if (e.getDirection() == Direction.NULL) {

// return true;

// }

// return false;

// }

**/\*\***

**\* Add a floor to the destination list.<br>**

**\***

**\* PRE: The floorID does not already have an active call button for this direction<br>**

**\* POST: The floorID is added to the list of floors requesting service<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param callBtn is the call button requesting service**

**\*/**

**public void addDestination(CallButton callBtn)**

{

requestList.add(callBtn);

callBtn.activate();

}

}

**DIRECTION (ENUM)**

public enum Direction {

NULL, // Indicates a stationary state

UP, // Indicates an upward direction

DOWN // Indicates a downward direction

}

**DOORTYPE (ENUM)**

public enum DoorType {

OUTER, // Indicates a door attached to a floor

INNER // Indicates a door belonging to an elevator

}

**ELEVATOR**

**/\*\***

**\* An elevator object that moves between floors and maintains a destination list**

**\* of what floors it needs to service.**

**\* @author MattCasiro**

**\* Created: 23 May 2016**

**\*/**

**public class Elevator**

{

private static final int MAX\_FLRS = 999;

private static final int MIN\_FLRS = 2;

private static final int MIN\_SUB\_FLRS = 0;

private boolean hasThirteen, hasZero, hasArrived;

private int elevatorID, shaftID, currentFloor, destination, topFloor, bottomFloor;

private int offset; // Offset is used to account for sub-floors when returning values

private static int count = 0; // ID's to track multiple elevators in larger system

private Door innerDoor;

private FloorButton[] floorButtons;

private DestinationList destinationList;

private Direction direction;

**/\*\***

**\* Instantiate an elevator object with only above-ground floors.<br>**

**\***

**\* Pre: A system exists to control the elevator<br>**

**\* Post: An elevator object was created and set to be on the first floor<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param numFloors is a positive number greater than one that is the number of above-ground floors the elevator can access**

**\* @param shaftID is the shaft in which the elevator operates, or 0 in a single elevator system**

**\* @param hasZero is true if the first floor is floor 0, or false if the first floor is floor 1**

**\* @param hasThirteen is True if the system has a 13th floor, otherwise False**

**\*/**

**public Elevator(int numFloors, int shaftID, boolean hasZero, boolean hasThirteen) throws IllegalArgumentException**

{

if (MIN\_FLRS > numFloors || numFloors > MAX\_FLRS)

{

throw new IllegalArgumentException();

}

direction = Direction.NULL;

this.hasZero = hasZero;

this.hasThirteen = hasThirteen;

elevatorID = getNextID();

currentFloor = (hasZero ? 0 : 1) + offset;

this.shaftID = shaftID;

this.topFloor = numFloors + (hasZero ? -1 : 0) + (hasThirteen ? 0 : 1);

this.offset = 0;

bottomFloor = (hasZero ? 0 : 1);

innerDoor = new Door();

destinationList = new DestinationList(numFloors);

destination = bottomFloor - 1;

try

{

setFloorButtons();

} catch (IllegalStateException ex) {

System.out.println("Illegal state reached when created elevator! Floors not instantiated.");

System.out.println(ex);

}

}

**/\*\***

**\* Instantiate an elevator object with above and below ground floors.<br>**

**\***

**\* Pre: A system exists to control the elevator<br>**

**\* Post: An elevator object was created and set to be on the first floor<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param numFloors is a positive number greater than one that is the number of above-ground floors the elevator can access**

**\* @param shaftID is the shaft in which the elevator operates, or 0 in a single elevator system**

**\* @param numSubFloors is a number greater than or equal to zero that is the number of below-ground floors the elevator can access**

**\* @param hasZero hasZero is true if the first floor is floor 0, or false if the first floor is floor 1**

**\* @param hasThirteen is True if the system has a 13th floor, otherwise False**

**\*/**

**public Elevator(int numFloors, int numSubFloors, int shaftID, boolean hasZero, boolean hasThirteen) throws** IllegalArgumentException

{

if (MIN\_FLRS > numFloors || MIN\_SUB\_FLRS > numSubFloors ||

numFloors > MAX\_FLRS || numSubFloors > MAX\_FLRS)

{

throw new IllegalArgumentException();

}

direction = Direction.NULL;

this.hasZero = hasZero;

this.hasThirteen = hasThirteen;

elevatorID = getNextID();

currentFloor = 1 + offset;

this.shaftID = shaftID;

this.topFloor = numFloors + (hasZero ? -1 : 0) + (hasThirteen ? 0 : 1);

this.offset = numSubFloors;

bottomFloor = -offset;

innerDoor = new Door();

destinationList = new DestinationList(numFloors + numSubFloors);

try

{

setFloorButtons();

} catch (IllegalStateException ex) {

System.out.println("Illegal state reached when creating elevator! Floors not instantiated.");

System.out.println(ex);

}

}

**/\*\***

**\* Create an array of FloorButton objects representing the**

**\* floors the elevator has access to.**

**\*/**

**private void setFloorButtons()**

{

int tmp = topFloor + offset;

// Add one to account for ignoring floor zero

floorButtons = new FloorButton[tmp + 1];

for (int i = 0, j = offset; i <= tmp; i++)

{

// Skip null floors

if ((!hasZero && i - offset == 0) || (!hasThirteen && i - offset == 13)) continue;

// Generate labels for each floor for use in GUI applications, where

// the pre-fix letter indicates an above-ground (F)loor or (B)asement floor

String name;

if (j > 0)

{

if (j < 10) name = "B:00";

else if (j < 100) name = "B:0";

else name = "B:";

floorButtons[i] = new FloorButton(-j, name + j);

j--;

} else {

if (i < 10) name = "F:00";

else if (i < 100) name = "F:0";

else name = "F:";

floorButtons[i] = new FloorButton(i-offset, name + (i-offset));

}

}

}

**/\*\***

**\* Increment elevator count and return an ID for an elevator.**

**\*/**

**private int getNextID()**

{

return (++count) + 100;

}

**/\*\***

**\* Get the state of the button for a specific floor.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param floor is the reference for which button to check**

**\* @return true if the button is active, false if the button is inactive**

**\*/**

**public boolean getButtonState(int floor)**

{

return floorButtons[floor].getButtonState();

}

**/\*\***

**\* Get the state of the light for a button for a specific floor.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param floor is the reference for which button to check**

**\* @return true if the light is active, false if the light is inactive**

**\*/**

**public boolean getButtonLightState(int floor)**

{

return floorButtons[floor].getLightState();

}

**/\*\***

**\* Get the state of the elevator's door.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @return true if the door is open, false if the door is closed**

**\*/**

**public boolean getDoorState()**

{

return innerDoor.isOpen();

}

**/\*\***

**\* Get the current number of elevators created.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @return the current number of elevators**

**\*/**

**public int getElevatorCount()**

{

return count;

}

**/\*\***

**\* Get the ID of the elevator object.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @return the elevator ID as an integer**

**\*/**

**public int getElevatorID()**

{

return elevatorID;

}

**/\*\***

**\* Get the shaft ID in which the elevator operates.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @return the ID number for the shaft the elevator was assigned to**

**\*/**

**public int getShaftID()**

{

return shaftID;

}

**/\*\***

**\* Get the current floor the elevator is on.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @return the current floor as an integer**

**\*/**

**public int getCurrentFloor()**

{

return currentFloor;

}

**/\*\***

**\* Get the top floor the elevator has access to.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @return the top floor as an integer**

**\*/**

**public int getTopFloor()**

{

return topFloor;

}

**/\*\***

**\* Get the lowest floor the elevator has access to.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @return the lowest floor as an integer (first floor is 1, basements are**

**\* reverse indexed starting at -1 and descending, floor zero never exists)**

**\*/**

**public int getBottomFloor()**

{

return bottomFloor;

}

**/\*\***

**\* Get the current destination for the elevator.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @return the next location the elevator will stop at given it's current state, as an integer**

**\*/**

**public int getDestination()**

{

return destination;

}

**/\*\***

**\* Get the current direction the elevator is traveling in.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @return 1 if the elevator is moving up, 0 if it is stationary, and -1 if it is moving down**

**\*/**

**public Direction getDirection()**

{

return direction;

}

**/\*\***

**\* Check if the elevator has arrived at the set destination.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @return true if the elevator has just arrived, false if not**

**\*/**

**public boolean isArrived()**

{

return hasArrived;

}

**/\*\***

**\* Adds a floor number to the destination list for this elevator.<br>**

**\***

**\* PRE: The request was generated by a floor button<br>**

**\* POST: The floor is added to the elevator's list of destinations, or the**

**\* elevator signals it's arrival if the floor is the current floor, the**

**\* floor button is activated and the button lit<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param floorID is a floor that is accessible by the elevator**

**\* @throws IllegalArgumentException if the floorID is invalid**

**\*/**

**public void selectFloor(int floorID) throws IllegalArgumentException**

{

innerDoor.close();

if (floorID > topFloor || bottomFloor > floorID)

{

throw new IllegalArgumentException();

}

floorButtons[floorID].activate();

// Signal arrived if the current floor is selected

if (floorID == currentFloor)

{

arrived();

// If the elevator has no destination, set the floorID as the destination

} else if (destination == bottomFloor - 1) {

destination = floorID;

// If the selected floor should be the next destination, put the current

// destination back in the destination list and use the selected floor

// as the new next destination

} else if (direction == Direction.UP && floorID > currentFloor && floorID < destination) {

destinationList.addDestination(destination, currentFloor, Direction.NULL);

destination = floorID;

} else if (direction == Direction.DOWN && floorID < currentFloor && floorID > destination) {

destinationList.addDestination(destination, currentFloor, Direction.NULL);

destination = floorID;

} else {

destinationList.addDestination(floorID, currentFloor, Direction.NULL);

}

// If the elevator is not moving, start it in the direction of the latest request

if (direction == Direction.NULL)

{

direction = (destination > currentFloor ? Direction.UP : Direction.DOWN);

}

}

**/\*\***

**\* Adds a floor number to the destination list for this elevator.<br>**

**\***

**\* PRE: The request was generated by a call button<br>**

**\* POST: The floor is added to the elevator's list of destinations, or the**

**\* elevator signals it's arrival if the floor is the current floor<br**

**\* Cleanup: N/A<br>**

**\***

**\* @param callBtn is the call button making the elevator service request**

**\*/**

**public void addServiceRequest(CallButton callBtn) throws IllegalArgumentException**

{

int floorID = callBtn.getFloorID();

Direction btnDirection = callBtn.getDirection();

if (floorID > topFloor || floorID < bottomFloor)

{

throw new IllegalArgumentException();

}

// Signal arrived if the current floor is selected

if (floorID == currentFloor)

{

arrived();

// If the elevator has no destination, set the floorID as the destination

} else if (destination < bottomFloor) {

destination = floorID;

direction = (destination > currentFloor ? Direction.UP : Direction.DOWN);

// If the selected floor should be the next destination, put the current

// destination back in the destination list and use the selected floor

// as the new next destination

} else if (this.direction == Direction.UP && floorID > currentFloor && floorID < destination && btnDirection == Direction.UP) {

destinationList.addDestination(destination, currentFloor, Direction.NULL);

destination = floorID;

} else if (this.direction == Direction.DOWN && floorID < currentFloor && floorID > destination && btnDirection == Direction.DOWN) {

destinationList.addDestination(destination, currentFloor, Direction.NULL);

destination = floorID;

} else {

destinationList.addDestination(floorID, currentFloor, btnDirection);

}

}

**/\*\***

**\* Conduct a single time-step event for an elevator.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: The elevator moves as required and the state is updated to represent**

**\* any required changes<br>**

**\* Cleanup: N/A<br>**

**\*/**

**public void tick() throws IllegalStateException**

{

hasArrived = false;

// Close door before moving, if open

if (innerDoor.isOpen()) innerDoor.close();

// Move in the current direction for the elevator

switch (direction) {

case NULL:

break;

case UP:

incrementFloor();

System.out.println("Moved to: "+currentFloor);

break;

case DOWN:

decrementFloor();

System.out.println("Moved to: "+currentFloor);

break;

default:

throw new IllegalStateException();

}

// Check if elevator is at next destination

if (currentFloor == destination)

{

System.out.println("arriving at " + destination);

arrived();

}

// Update direction for next destination

if (destination < bottomFloor)

{

direction = Direction.NULL;

} else if (destination > currentFloor) {

System.out.println("Setting direction: Up");

direction = Direction.UP;

} else if (destination < currentFloor) {

System.out.println("Setting direction: Down");

direction = Direction.DOWN;

} else {

throw new IllegalStateException();

}

}

**/\*\***

**\* Move the elevator up one floor.**

**\***

**\* PRE: The elevator is not on the top floor**

**\* POST: The elevator ascends one floor**

**\***

**\* @throws IllegalStateException if the elevator tries to ascend above the highest floor**

**\*/**

**private void incrementFloor() throws IllegalStateException**

{

int tmpFloor = currentFloor;

currentFloor++;

// Move past null floors

if ((!hasZero && currentFloor == 0) || (!hasThirteen && currentFloor == 13))

{

currentFloor++;

}

// Prevent crashing in to the top of the shaft

if (currentFloor > topFloor)

{

currentFloor = tmpFloor;

throw new IllegalStateException();

}

}

**/\*\***

**\* Move the elevator down one floor.**

**\***

**\* PRE: The elevator is not on the bottom floor**

**\* POST: The elevator descends one floor**

**\***

**\* @throws IllegalStateException if the elevator tries to descend below the lowest floor**

**\*/**

**private void decrementFloor() throws IllegalStateException**

{

int tmpFloor = currentFloor;

currentFloor--;

// Move past null floors

if ((!hasZero && currentFloor == 0) || (!hasThirteen && currentFloor == 13))

{

currentFloor--;

}

// Prevent crashing in to the bottom of the shaft

if (currentFloor < -offset)

{

currentFloor = tmpFloor;

throw new IllegalStateException();

}

}

**/\*\***

**\* Play a chime noise and output a chime message to console.**

**\*/**

**private void playChime()**

{

Toolkit.getDefaultToolkit().beep();

System.out.println("Elevator " + elevatorID + " chimed!");

}

**/\*\***

**\* Adjust the elevator state to account for arriving at the given destination**

**\*/**

**private void arrived()**

{

if(!hasArrived) {

floorButtons[currentFloor].deactivate();

innerDoor.open();

playChime();

hasArrived = true;

do {

destination = destinationList.getNextDestination(bottomFloor, direction).floorID;

} while (destination == currentFloor);

}

}

**/\*\***

**\* Get the string representation of the elevator's state.**

**\***

**\* PRE: N/A**

**\* POST: N/A**

**\* Cleanup: N/A**

**\***

**\* @return the elevator's state as a string**

**\*/**

**@Override**

**public String toString()**

{

return "Elvator State:"+

"\n\tID:\t"+elevatorID+

"\n\tLOC:\t"+currentFloor+

"\n\tDEST:\t"+destination;

}

}

**ELEVATOR SYSTEM**

**/\*\***

**\* An elevator system controller.**

**\* @author MattCasiro**

**\* Created: 23 May 2016**

**\*/**

**public class ElevatorSystem {**

public static final int MAX\_FLRS = 999;

public static final int MIN\_FLRS = 2;

public static final int MIN\_SUB\_FLRS = 0;

private boolean hasThirteen, hasZero;

private int topFloor, bottomFloor, numShafts;

private int offset; // Offset is used to account for sub-floors when returning values

private CallRequestList upRequestList;

private CallRequestList downRequestList;

private Door[][] outerDoors;

private Stack<Door> openDoors;

private CallButton[] upButtonList;

private CallButton[] downButtonList;

private Elevator[] elevators;

**/\*\***

**\* Instantiate an elevator system with a given number of floors and elevator shafts.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param numFloors is the number of above-ground floors the elevator services**

**\* @param numSubFloors is the number of below-ground floors the elevator services**

**\* @param hasZero is true if the floors are zero indexed, false if there is no floor zero**

**\* @param hasThirteen indicates whether or not there is a 13th floor**

**\* @param numShafts is the number of separate elevator shafts in the system**

**\* @throws IllegalArgumentException if numFloors or numSubFloors is out of the ranges:<br>**

**\* (MIN\_FLRS <= numFloors <= MAX\_FLRS) and (MIN\_SUB\_FLRS <= numSubFloors <= MAX\_FLOORS)**

**\*/**

**public ElevatorSystem(int numFloors, int numSubFloors, boolean hasZero, boolean hasThirteen, int numShafts)**

throws IllegalArgumentException

{

if (MIN\_FLRS > numFloors || MIN\_SUB\_FLRS > numSubFloors ||

numFloors > MAX\_FLRS || numSubFloors > MAX\_FLRS)

{

throw new IllegalArgumentException();

}

this.hasZero = hasZero;

this.hasThirteen = hasThirteen;

this.numShafts = numShafts;

// Add a floor if skipping thirteen unless floor count is under 13

this.topFloor = numFloors + (hasZero ? 0 : -1) + (hasThirteen ? 0 : 1);

this.offset = numSubFloors;

this.bottomFloor = (offset != 0) ? -offset : (hasZero ? 0 : 1);

upRequestList = new CallRequestList(Direction.UP);

downRequestList = new CallRequestList(Direction.DOWN);

// Add one to account for ignoring floor zero

upButtonList = new CallButton[topFloor + offset + 1];

downButtonList = new CallButton[topFloor + offset + 1];

outerDoors = new Door[topFloor + offset + 1][numShafts];

// setCallButtons(upButtonList, Direction.UP);

// setCallButtons(downButtonList, Direction.DOWN);

// createDoors();

createFloors();

createElevators(numFloors, numSubFloors);

openDoors = new Stack<>();

}

**/\*\***

**\* Initialize all objects required for a floor to function, including**

**\* doors, up buttons, and down buttons**

**\*/**

**private void createFloors()**

{

int tmp = topFloor + offset;

for (int i = 0, j = offset; i <= tmp; i++)

{

// Skip floor zero (and thirteen if required)

if ((!hasZero && i - offset == 0) || (!hasThirteen && i - offset == 13)) continue;

// Generate labels for each floor for use in GUI applications

// the pre-fix letter indicates an above-ground (F)loor or (B)asement floor

String name;

// Create subfloors

if (j > 0)

{

if (j < 10) name = "B:00";

else if (j < 100) name = "B:0";

else name = "B:";

// Create up button

upButtonList[i] = new CallButton(-j, name + j, Direction.UP);

// Create down buttons

downButtonList[i] = new CallButton(-j, name + j, Direction.DOWN);

// Create outer doors for each shaft

for (int s = 0; s < numShafts; s++)

{

outerDoors[i][s] = new Door(-j, name, s);

}

// Create Elevator

j--;

// Create main floors

} else {

if (i < 10) name = "F:00";

else if (i < 100) name = "F:0";

else name = "F:";

// Create up bottons

upButtonList[i] = new CallButton(i - offset, name + i, Direction.UP);

// Create down buttons

downButtonList[i] = new CallButton(i-offset, name + (i-offset), Direction.DOWN);

for (int s = 0; s < numShafts; s++)

{

outerDoors[i][s] = new Door(-j, name, s);

}

}

}

}

**/\*\***

**\* Create elevator objects for each shaft**

**\*/**

**private void createElevators(int numFloors, int numSubFloors)**

{

elevators = new Elevator[numShafts];

for (int s = 0; s < numShafts; s++)

{

elevators[s] = new Elevator(numFloors, numSubFloors, hasZero, hasThirteen);

}

}

**/\*\***

**\* Get the state of the door for a given floor -> shaft combination.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param floor is the floor the door is on**

**\* @param shaft is the shaft the door belongs to**

**\* @return true if the door is open, false if the door is closed**

**\*/**

**public boolean isFloorDoorOpen(int floor, int shaft) throws IllegalArgumentException**

{

if (bottomFloor > floor || floor > topFloor ||

shaft < 0 || shaft > elevators.length - 1)

{

throw new IllegalArgumentException();

}

return outerDoors[floor][shaft].isOpen();

}

**/\*\***

**\* Get the state of a given call button.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param floor is the floor the button is on**

**\* @param direction indicates if you are checking the UP button or the DOWN button**

**\* @return true if the button is active, false if the button is inactive**

**\*/**

**public boolean isCallButtonActive(int floor, Direction direction)**

{

return (direction == Direction.UP ? upButtonList[floor].getButtonState() : downButtonList[floor].getButtonState());

}

**/\*\***

**\* Get the state of a given call button's light.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param floor is the floor the button is on**

**\* @param direction indicates if you are checking the UP button or the DOWN button**

**\* @return true if the light is on, false if the light is off**

**\*/**

**public boolean isCallButtonLightLit(int floor, Direction direction)**

{

return (direction == Direction.UP ? upButtonList[floor].getLightState() : downButtonList[floor].getLightState());

}

**/\*\***

**\* Get the state of a given elevator's given floor button.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param floor is the button in the elevator to check**

**\* @param shaft is the shaft the elevator to check is in**

**\* @return true if the button is active, false if the button is inactive**

**\*/**

**public boolean isElevatorButtonActive(int floor, int shaft)**

{

return elevators[shaft].getButtonState(floor);

}

**/\*\***

**\* Get the state of a given elevator's given button's light.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param floor is the button in the elevator to check**

**\* @param shaft is the shaft the elevator to check is in**

**\* @return true if the light is on, false if the light is off**

**\*/**

**public boolean isElevatorButtonLightLit(int floor, int shaft)**

{

return elevators[shaft].getButtonLightState(floor);

}

**/\*\***

**\* Get the state of a given elevator's door.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param shaft is the shaft the elevator to check is in**

**\* @return true if the door is open, false otherwise**

**\*/**

**public boolean isElevatorDoorOpen(int shaft)**

{

return elevators[shaft].getDoorState();

}

**/\*\***

**\* Get the location of a given elevator.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: N/A<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param shaft is the shaft in which the elevator resides**

**\* @return the floor the elevator is on as an integer**

**\*/**

**public int getElevatorLocation(int shaft)**

{

return elevators[shaft].getCurrentFloor();

}

**/\*\***

**\* Call an elevator to service the specified floor.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: The service request is logged for assignment by the system<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param floor is the floor requesting the service**

**\* @param direction is the intended direction of travel for the service request**

**\*/**

**public void callElevator(int floor, Direction direction) throws IllegalArgumentException**

{

if (bottomFloor > floor || floor > topFloor)

{

throw new IllegalArgumentException();

}

switch (direction) {

case UP:

upRequestList.addDestination(upButtonList[floor]);

break;

case DOWN:

downRequestList.addDestination(downButtonList[floor]);

break;

default:

throw new IllegalArgumentException();

}

}

**/\*\***

**\* Record the selection of a destination from inside an elevator.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: The system updates the elevator with the given destination<br>**

**\* Cleanup: N/A<br>**

**\***

**\* @param floor is the floor the user wishes to stop at**

**\* @param shaft is the shaft in which the elevator resides**

**\* @throws IllegalArgumentException if floor or shaft are invalid**

**\*/**

**public void selectFloor(int floor, int shaft) throws IllegalArgumentException**

{

try {

elevators[shaft].selectFloor(floor);

outerDoors[elevators[shaft].getCurrentFloor()][shaft].close();

} catch (ArrayIndexOutOfBoundsException ex) {

throw new IllegalArgumentException();

}

}

**/\*\***

**\* Conduct a single time-event for the elevator system.<br>**

**\***

**\* PRE: N/A<br>**

**\* POST: All open floor doors are closed, all elevators are assigned any**

**\* possible destinations, all elevator time events are called, and arrival**

**\* states checked<br>**

**\* Cleanup: N/A<br>**

**\***

**\*/**

**public void tick()**

{

// Close any open doors

while (!openDoors.empty())

{

openDoors.pop().close();

}

// Add destinations to elevators

upRequestList.getDestinations(elevators[0]);

downRequestList.getDestinations(elevators[0]);

// Tick all elevators and check if they have arrived at a destination

for (Elevator e : elevators)

{

e.tick();

System.out.println(e);

if (e.isArrived())

{

elevatorArrived(e.getCurrentFloor(), e.getShaftID());

}

}

}

**/\*\***

**\* Adjust door and button states when an elevator arrives**

**\*/**

**private void elevatorArrived(int floorID, int shaftID) {**

outerDoors[floorID][shaftID].open();

openDoors.push(outerDoors[floorID][0]);

upButtonList[floorID].deactivate();

downButtonList[floorID].deactivate();

}

}