class Stock {
    private final String name_;  
    final private double base_;  
    private int ref_;  
    private double price_;  

    //@ INV: base_ >= 0 && ref_ >= 0 && price_ == (base_ + 0.1 * ref_)
    Stock(String name, double base) throws WrongStockBasePriceException {  
        if (base < 0)  
            throw new WrongStockBasePriceException(base);  
        base_ = base;  
        ref_ = 0;  
        price_ = base;  
        name_ = name;  
    }
    public synchronized double getPrice() { return price_; }  
    public synchronized void incRef() { ref_++; resetPrice(); }  
    public synchronized void decRef() {  
        if (ref_ > 0) {  
            ref--;  
            resetPrice();  
        }  
    }
}

private synchronized void resetPrice() {  
    price_ = base_ + 0.1 * ref_;  
}
```java
class BankAccount {
    private final int ID_;  
    private double savings_; 
    private double maxOverDraft_; 
    private HashMap<Stock, Integer> stocks_; 

    //INV: savings_ + maxOverDraft_ > 0

    BankAccount(int ID, double amount, double maxOverDraft) { 
        if (ID < 1 || maxOverDraft < 0 || amount < maxOverDraft) 
            throw new WrongAccountDataException();
        ID_ = ID;  savings_ = amount;  maxOverDraft_ = maxOverDraft;
        stocks_ = new HashMap<Stock, Integer>();
    }

    public synchronized void incSaving(double amount) { savings_ += amount; }
    public synchronized void decSaving(double amount) { savings_ -= amount; }
    public synchronized void buyStock(Stock stock) { 
        if ((savings_ + maxOverDraft_) >= stock.getPrice()) { 
            Integer num = stocks_.get(stock);
            if (num == null) 
                stocks_.put(stock, new Integer(1)); 
            else 
                stocks_.put(stock, new Integer(num.intValue() + 1));
            decSaving(stock.getPrice());
            stock.incRef();
        } 
    }

    public synchronized void sellStock(Stock stock) { 
        Integer num = stocks_.get(stock); 
```
if (num != null) {
    if (num.intValue() > 1)
        stocks_.put(stock, num.intValue() - 1);
    else
        stocks_.remove(stock);
    incSaving(stock.getPrice());
    stock.decRef();
}
}

interface Command {
    public void apply(SynchMap<Integer, BankAccount> accounts,
                      SynchMap<String, Stock> stocks);
}

class Dealer extends Thread {

    SynchMap<String, Stock> stocks_;
    SynchMap<Integer, BankAccount> accounts_;
    SynchRandomCollection <Command> commands_;

    Dealer(SynchMap<Integer, BankAccount> accounts,
           SynchMap<String, Stock> stocks,
           SynchRandomCollection <Command> commands)  
    { stocks_ = stocks; accounts_ = accounts; commands_ = commands; }

    public void run() {
        while (true) {
            Command command = commands_.remove();
            command.apply(accounts_.stocks_);  
        }
    }
}
class BuyCommand implements Command {
    protected int accountId_;  
    protected String stockName_;  
    BuyCommand(int accountId, String stockName) {
        accountId_ = new Integer(accountId);
        stockName_ = stockName;
    }

    public void apply(SynchMap<Integer,BankAccount> accounts, 
                        SynchMap<String,Stock> stocks) {
        BankAccount account = accounts.get(accountId_);
        Stock stock = stocks.get(stockName_);
        if (account != null && stock != null)
            account.buyStock(stock);
    }
}

class SellCommand implements Command {
    protected int accountId_;  
    protected String stockName_;  
    SellCommand(int accountId, String stockName) {
        accountId_ = new Integer(accountId);
        stockName_ = stockName;
    }

    public void apply(SynchMap<Integer,BankAccount> accounts, 
                        SynchMap<String,Stock> stocks) {
        BankAccount account = accounts.get(accountId_);
        Stock stock = stocks.get(stockName_);
        if (account != null && stock != null)
            account.sellStock(stock);
    }
}
In a file named "BankAccount Stock" (8 lines total), the following text is extracted:

1. A BankAccount class is defined with methods for depositing and withdrawing funds.
2. The class also includes methods for checking the balance and transferring funds between accounts.
3. The class is instantiated with a balance of 1000 and methods are called to demonstrate functionality.

In a file named "Elevator" (30 lines total), the following text is extracted:

1. The Elevator class is defined with methods for going up, down, and being idle.
2. The class also includes methods for handling requests and managing the elevator's trajectory.
3. The class is instantiated with starting and ending floors, and methods are called to demonstrate functionality.

The Elevator class is used to demonstrate how an elevator system can manage requests and move between floors.

```cpp
#include <set>
#include <vector>
#include <iostream>

enum Direction { UP, DOWN, NONE };  
enum RequestType { GO, CALL };  

class Request; 
typedef std::vector<const Request*> Requests; 

class Trajectory 
{  
public:  
typedef std::set<int> Stops; 
Trajectory(Direction direction) : direction_(direction) {}  
Direction getDirection() const { return direction_; } 
}; 
```
// While in currentFloor, add a new stop request on the trajectory
// The request is ignored if it cannot be added to the trajectory
// Return true if the trajectory is modified
bool addStop(int currentFloor, int floor) {
    if (direction_ == DOWN) {
        if (floor < currentFloor) {
            stops_.insert(floor);
            return true;
        }
    } else if (direction_ == UP) {
        if (floor > currentFloor) {
            stops_.insert(floor);
            return true;
        }
    }
    return false;
}

// Is the trajectory empty?
bool empty() const { return stops_.empty(); }

// What is the next stop to which we want to stop?
int nextStop(int currentFloor) const {
    if (direction_ == UP)
        return (*stops_.lower_bound(currentFloor));
    else if (direction_ == DOWN)
        return (*stops_.upper_bound(currentFloor));
    else
        return -1;
}

// Remove element to which | points from the set
void removeStop(int floor) {
    stops_.erase(floor);
}

private:
    const Direction direction_; 
    Stops stops_; //Sorted list of floors on which the trajectory will stop.
// A request sent to the elevator:
// - Goto a certain floor
// - Call the elevator from a certain floor to go up or down

class Request {
public:
  Request(RequestType rt, int floor) : type_(rt), floor_(floor), direction_(NONE) {}
  Request(RequestType rt, int floor, Direction direction) : type_(rt), floor_(floor), direction_(direction) {}
  int getFloor() const {return floor_;}
  Direction getDirection() const {return direction_;}
  RequestType getType() const {return type_;}
private:
  const int floor_;
  const Direction direction_;
  const RequestType type_;
};

class Elevator {
public:
  enum State {GOING_UP, GOING_DOWN, IDLE};
  Elevator(int numFloors) : numFloors_(numFloors), currentFloor_(0), state_(IDLE), currentTraj_(UP) {}

  // Remember that a request has been sent to the elevator
  // If the request is compatible with the current trajectory,
  // add it for immediate handling
  // else remember it for one of the next trajectories
  void addRequest(const Request& r) {
    if (canAddRequestToTraj(r)) {
      currentTraj_.addStop(currentFloor_, r.getFloor());
    } else {
      pendingRequests_.push_back(r);
    }
  }

  // Perform the requests, return after an adjacent floor has been
  // reached or if there are no requests to service.
void handleRequests(Requests* newRequests);

private:
    const int numFloors_;
    int currentFloor_;
    State state_;
    Trajectory currentTraj_;
    Requests pendingRequests_;

    // Can we reach target from current by going in direction direction?
    bool follow(int targetFloor, int currentFloor, Direction direction) {
        if (direction == UP)
            return (targetFloor > currentFloor);
        else
            return (targetFloor < currentFloor);
    }

    // Can a request be added to the current trajectory?
    bool canAddRequestToTraj(const Request& r) {
        if ((r.getType()==CALL && r.getDirection()==currentTraj_.getDirection()) || r.getType() == GO)
            return follow(r.getFloor(), currentFloor_, currentTraj_.getDirection());
        else
            return false;
    }
};

void testEmptyTraj1() {
    Trajectory t1(UP);
    t1.addStop(1, 3);
    if (t1.empty()) {
        std::cerr << "Error: empty trajectory after addStop" << std::endl;
    } else {
        std::cout << "EmptyTraj1 passed" << std::endl;
    }
}
void testEmptyTraj2() {
    Trajectory t1(UP);
    if (!t1.empty()) {
        std::cerr << "Error: trajectory not empty when constructed" << std::endl;
    } else {
        std::cout << "EmptyTraj2 passed" << std::endl;
    }
}

// Get random requests arriving between clock-1 and clock
Requests* getRequests(int clock) {  
    Requests* r = new Requests();

    switch (clock)
    {
    case 1:
        r->push_back(new Request(CALL,1,UP));
        break;
    case 2:
        r->push_back(new Request(CALL,2,DOWN));
        break;
    case 3:
        r->push_back(new Request(CALL,3,DOWN));
        break;
    case 4:
        r->push_back(new Request(UP));
        break;
    default:
        break;
    }
    return r;
}
```java
case 3:
    v->push_back(new Request(GO,2));
    v->push_back(new Request(CALL,2,DOWN));
    break;
}

return v;
}

int main() {
    Elevator elev(3); // Elevator moves between floors 0 and 3 included
    int clock = 0; // The clock of the simulation.

    // Main simulation loop
    while (true) {
        clock++; // New round
        // Get all requests until time clock, and handle the requests
        elev.handleRequests(getRequests(clock));
    }
    return 0;
}
```

```java
class Client1 {
    public static void main(String args[]) {
        try {
            Socket socket = new Socket("tapuz",1300);
            {
                //do something
            }
            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```
class Client2 {
    public static void main(String args[]) {
        try {
            DatagramSocket socket = new DatagramSocket();
            // do something
            socket.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

Connection (connection) and transport (transport layer) are terms used to describe the relationship between a client and a server in a networking context.

The method `isAcceptable()` tests whether this key's channel is ready to accept a new socket connection.

The method `isConnectable()` tests whether this key's channel is ready, so that invoking `finishConnect()` does not block.

| static int | OP_ACCEPT | Operation-set bit for socket-accept operations. |
| static int | OP_CONNECT | Operation-set bit for socket-connect operations. |
| static int | OP_READ   | Operation-set bit for read operations. |
| static int | OP_WRITE  | Operation-set bit for write operations |

SelectionKey

Methode `isAcceptable()` תדעו של `isConnectable` של המחלקה `SelectionKey`.
### boolean isReadable()
Tests whether this key's channel is ready for reading.

### boolean isWritable()
Tests whether this key's channel is ready for writing.

### static SocketChannel open()
Opens a socket channel.

### SelectorProvider configureBlocking(boolean block)
Adjusts this channel's blocking mode.

### SelectionKey register(Selector sel, int ops, Object att)
 Registers this channel with the given selector, returning a selection key.

### abstract void cancel()
Requests that the registration of this key's channel with its selector be cancelled.

### abstract boolean connect(InetSocketAddress remote)
Connects this channel's socket. Return true if a connection was established, false if this channel is in non-blocking mode and the connection operation is in progress.

### abstract boolean finishConnect()
Finishes the process of connecting a socket channel. Return true if, and only if, this channel's socket is now connected.