```java
public class Main {
    // a simple example of how you might use the parking app
    public static void main(String[] args) {
        Parkzilla p = new Parkzilla();
        p.addLot("Lot 1", 100, 100);
        p.addLot("Lot 2", 50, 200);

        // add a customer
        Customer c = new PrivilegedCustomer("Alice");
        p.addCustomer(c);

        // park in an empty slot
        final int PARK_TIME = 30;
        final String PARKING_LOT = "Lot 1";
        Stall s = p.findEmptyStall(PARKING_LOT);
        if (s != null) {
            if (c.isAllowedToPark(PARKING_LOT, new Time(PARK_TIME))) {
                c.park(s, PARK_TIME);
            } else {
                System.out.println("Customer is not permitted to park here right now.");
            }
        } else {
            System.out.println("Lot is full");
        }

        // leave the parking spot
        c.unpark();

        // pay for a ticket if there is one
        c.payViolations(0);

        // incomplete examples using the other types of customers (may not be of interest)
        Set<String> lots = new HashSet<>();
        lots.add("Lot 1");
        c = new LotRestrictedCustomer ("Bob", lots);
        p.addCustomer(c);
        s = p.findEmptyStall("Lot 2");
        c.park(s, 30);
        assert(!c.isParked());

        Set<Integer> days = new HashSet<>();
        days.add(Calendar.MONDAY);
        c = new DayRestrictedCustomer ("Cathy", days);
        p.addCustomer(c);

        c = new UnpaidTicketRestrictedCustomer("Doug");
        p.addCustomer(c);

        p.checkForViolations();
    }
}
```

```java
// Manage paid parking for multiple parking lots
public class Parkzilla {
    public final static int VIOLATION_AMOUNT = 100;
    private City city = new City();
    private List<Customer> customers = new ArrayList<>();

    // REQUIRES: numStalls > 0 and costPerMinute > 0
    // MODIFIES: this
    // EFFECTS: adds parking with name, number of parking stalls, and cost structure
    public void addLot(String name, int numStalls, int costPerMinute) {
        Lot lot = new Lot(name, numStalls, costPerMinute);
        city.addLot(lot);
    }

    // MODIFIES: this
    // EFFECTS: add new customer
    public void addCustomer(Customer customer) {
        customers.add(customer);
    }

    // REQUIRES: lot with name lotName has been added to city
    // EFFECTS: returns an empty parking stall or null if all stalls in lot are occupied
    public Stall findEmptyStall(String lotName) {
        Lot lot = city.getLot(lotName);
        return lot.findEmptyStall();
    }

    // MODIFIES: this
    // EFFECTS: iterates over all stalls to look for stalls occupied by customer
    //          whose time has expired and calls addViolation() for each of them
    public void checkForViolations() {
        for (Stall stall : city) {
            Customer customerInViolation = stall.isInViolation();
            if (customerInViolation!=null) {
                customerInViolation.addViolation();
            }
        }
    }
}
```

```java
// A parking violation
public class Violation {
    private Time time;

    public Violation() {
        time = new Time(0);
    }
}
```

```java
// Customers pay for and park in stalls and may have to pay fines if they park too long
public abstract class Customer {
    private String name;
    private List<Violation> violations = new ArrayList<>();
    private Stall parkedInStall;

    // EFFECTS: constructs a new customer with specified name
    public Customer(String name) {
        this.name = name;
    }

    // EFFECTS: returns true if customer is permitted in specified lot until endTime
    public abstract boolean isAllowedToPark(String lot, Time endTime);

    public boolean isParked() {
        return parkedInStall != null;
    }

    // EFFECTS: returns this customer's current number of unpaid violations
    public int getNumUnpaidViolations() {
        return violations.size();
    }

    // REQUIRES: that stall is not null and is available and
    //           that customer is allow to park in lot until endTime
    // MODIFIES: this and stall
    // EFFECTS: sets stall to be occupied by this customer and
    //          paid for next durationMinutes minutes
    public void park(Stall stall, int durationMinutes) {
        Time endTime = new Time(durationMinutes);
        if (isAllowedToPark(stall.getLot().getName(), endTime)) {
            charge(stall.getCost(durationMinutes));
            stall.setOccupied(this, endTime);
            parkedInStall = stall;
        }
    }

    // EFFECTS: removes customer from parking stall
    public void unpark() {
        parkedInStall = null;
    }

    // EFFECTS: charges customer by amount
    public void charge(int amount) {
        // implemention ommitted to save space
    }

    // MODIFIES: this
    // EFFECTS: increments customer's number of unpaid parking violations
    public void addViolation() {
        violations.add(new Violation());
    }

    // REQUIRES: numUnpaidVioltions >= numToPay
    // MODIFIES: this
    // EFFECTS: charges customer and reduces numUnpaidViolations by that amount
    public void payViolations(int numToPay) {
        charge(numToPay * Parkzilla.VIOLATION_AMOUNT);
        for (int i=0; i<numToPay; i++)
            violations.remove(0);
    }
}
```

```java
// A customer that can park anywhere at any time
public class PrivilegedCustomer extends Customer {

    public PrivilegedCustomer(String name) {
        super(name);
    }

    // EFFECTS: returns true if customer is permitted in specified lot until endTime
    @Override
    public boolean isAllowedToPark(String lot, Time endTime) {
        return true;
    }
}

// A customer that can only park in certain parking lots
public class LotRestrictedCustomer extends Customer {
    private Set<String> permittedLots;

    public LotRestrictedCustomer(String name, Set<String> permittedLots) {
        super(name);
        this.permittedLots = Collections.unmodifiableSet(permittedLots);
    }

    // EFFECTS: returns true if customer is permitted in specified lot until endTime
    @Override
    public boolean isAllowedToPark(String lot, Time endTime) {
        return permittedLots.contains (lot);
    }
}

// A customer than can only park on certain days
public class DayRestrictedCustomer extends Customer {
    private Set<Integer> permittedDays;

    public DayRestrictedCustomer(String name, Set<Integer> permittedDays) {
        super(name);
        this.permittedDays = Collections.unmodifiableSet(permittedDays);
    }

    // EFFECTS: returns true if customer is permitted in specified lot until endTime
    @Override
    public boolean isAllowedToPark(String lot, Time endTime) {
        return permittedDays.contains(endTime.getDayOfWeek());
    }

}

// A customer that can only park if she has no unpaid parking tickets
public class UnpaidTicketRestrictedCustomer extends Customer {

    public UnpaidTicketRestrictedCustomer(String name) {
        super(name);
    }

    // EFFECTS: returns true if customer is permitted in specified lot until endTime
    @Override
    public boolean isAllowedToPark(String lot, Time endTime) {
        return getNumUnpaidViolations() == 0;
    }
}
```

```java
// A city is just a collection of parking lots
// Iterating over a city means iterating over every stall in every parking lot in the city
public class City {
    private Map<String, Lot> lots = new HashMap<>();

    // MODIFIES: this
    // EFFECTS: adds lot to list of parking lots
    public void addLot(Lot lot) {
        lots.put(lot.getName(), lot);
    }

    // EFFECTS: returns parking lot with specified name or null if not found
    public Lot getLot(String name) {
        return lots.get(name);
    }
}
```

```java
// A parking lot is a collection of parking stalls
public class Lot  {
    private String name;
    private List<Stall> stalls;

    // REQUIRES: numStalls > 0 and costPerMinute > 0
    // EFFECTS: constructs a new parking lot
    public Lot(String name, int numStalls, int costPerMinute) {
        this.name = name;
        stalls = new ArrayList<>();
        for (int i=0; i<numStalls; i++) {
            stalls.add(new Stall(this, costPerMinute));
        }
    }

    // EFFECTS: returns name of parking lot
    public String getName() {
        return name;
    }

    // EFFECTS: returns an empty stall or null if lot is full
    public Stall findEmptyStall() {
        for (Stall stall : stalls)
            if (stall.isEmpty())
                return stall;
        return null;
    }
}
```

```java
// Encapsulates a single parking stall that is part of a particular parking lot
public class Stall {
    private Lot lot;
    private int costPerMinute;
    private Customer occupiedBy;
    private Time paidUntil;

    // REQUIRES: costPerMinute > 0
    // MODIFIES: this
    // EFFECTS: constructs stall with specified lot and cost structure
    public Stall(Lot lot, int costPerMinute) {
        this.lot = lot;
        this.costPerMinute = costPerMinute;
    }

    // EFFECTS: returns name of parking lot where stall is located
    public Lot getLot() {
        return lot;
    }

    // EFFECTS: returns cost for parking in stall for specified minutes
    public int getCost(int minutes) {
        return minutes * costPerMinute;
    }

    // EFFECTS: returns true if and only if the stall is empty
    public boolean isEmpty() {
        return occupiedBy == null;
    }

    // EFFECTS: returns true if stall is occupied beyond time the stall was paid for
    public Customer isInViolation() {
        if (occupiedBy != null && paidUntil.isBeforeNow())
            return occupiedBy;
        else
            return null;
    }

    // MODIFIES: this
    // EFFECTS: sets stall as occupied by "customer" and paid for until time "paidFor"
    public void setOccupied(Customer customer, Time paidUntil) {
        this.occupiedBy = customer;
        this.paidUntil  = paidUntil;
    }

    // EFFECTS: removes customer from this stall
    public void setEmpty() {
        occupiedBy = null;
    }
}
```

```java
// Encapsulates a specific time of day - YOU CAN IGNORE THE DETAILS THIS CLASS
public class Time {
    Calendar calendar;

    // EFFECTS: constructs a new time object whose time is minutesFromNow in the future
    public Time(int minutesFromNow) {
        calendar = Calendar.getInstance();
        calendar.add(Calendar.MINUTE, minutesFromNow);
    }

    // EFFECTS: returns day of week as integer 0..6 representing Sunday, Monday, ... Saturday
    public int getDayOfWeek() {
        return calendar.get(Calendar.DAY_OF_WEEK);
    }

    // EFFECTS: returns true if an only if this time is before the current time
    public boolean isBeforeNow() {
        return calendar.compareTo(Calendar.getInstance()) == -1;
    }
}
```