1. UML Class Diagram [10 marks]

Extract a complete UML class diagram (including associations) for all types in the code pack except for `Main`. Do not include any types from the Java library. Do not include fields, methods, or dependencies.

2. Sequence Diagram [10 marks]

Draw a sequence diagram starting with a call to `park` on an instance of `LotRestrictedCustomer`. Do not include calls to the Java library. Be sure that you include constructor calls, use a box for conditional statements, and include return arrows (values are not required). Include a legend if you abbreviate any names.

3. Java Basics [9 marks]

a) There is one line of the `Customer` class where you see the power of subtype polymorphism (aka type substitution). This line names a particular method, but when it executes it selects among a set of alternate implementations of that method depending on the type of the object on which it is invoked. Put its line number in the box to the right.

b) Consider the following code

```
PrivilegedCustomer a = new PrivilegedCustomer("Alice");
PrivilegedCustomer b = new PrivilegedCustomer("Alice");
PrivilegedCustomer c = b;
Customer d = b;
```

Indicate whether each of the following expressions evaluates to `true` or `false`.

Expression	Value (`true` or `false`)
`a == b`	
`b == c`	
`b == d`	

c) The following assertion will fail.

```
Set<Customer> cs = new HashSet<>();
cs.add(new PrivilegedCustomer("Alice"));
cs.add(new PrivilegedCustomer("Alice"));
assertEquals(cs.size(), 1);
```

Briefly describe what you would have to change for the assertion to succeed. List the names of all methods that you would add or change. And list the field(s) your new code would have to access. Do not give any other implementation details.

4. Writing Robust Code [14 marks]

Notice that the `Customer.park` is not robust. It is repeated here for convenience.

```
// 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;
    }
}
```

In this question you will modify `park` to make it robust. Notice that this code does actually check one of the requirements, though it does nothing if it fails. Your solution must remove both requirements by throwing exceptions instead.

a) Define two exceptions that `park` will throw: `StallNotAvailable`, an ***unchecked*** exception; and `CustomerNotAllowed`, a ***checked*** exception. Complete the following declarations of these two exceptions.

```
class StallNotAvailable extends _____________________________
```

```
class CustomerNotAllowed extends _____________________________
```

b) Rewrite the specification, signature, and implementation of `park` to make it robust using these two exceptions.

c) Rewrite this code from `Main` that calls `park`, repeated here for convenience, so that it has the same behaviour when it calls your new version of `park` but **without any `if` statements**.

```
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");
}
```

d) Complete the following unit test for your robust version of `park`. Assume that `@BeforeEach` does nothing, which means you need to place all initialization code in the test method itself. Write the test as we did in class; in particular, it may not throw exceptions and you may not use the JUnit "expected" clause (its fine if you don't know what this is).

```
@Test
void testWithOccupiedStall() {




}
```

5. Data Abstraction [7 marks]

Consider adding the following field to the `Parkzilla` class.

```
private Map<Customer, List<Violation>> allViolations = new HashMap<>();
```

Implement the following new `Parkzilla` method to maintain a customer-keyed, master list of parking violations. (Assume that the `Customer` class is implemented correctly for use as a hash-map key.)

```
// MODIFIES this
// EFFECTS adds violation to list of violations for customer stored in allViolations map
private void addViolation (Customer c, Violation v) {

}
```

6. Bidirectional Associations [8 marks]

The methods `Customer.unpark` and `Stall.setEmpty` do not correctly maintain the bidirectional association between a `Customer` and a `Stall`.

Provide a correct implementation for `Customer.unpark`.

```
// EFFECTS: removes customer from parking stall
public void unpark() {




}
```

Provide a correct implementation for `Stall.setEmpty`.

```
// EFFECTS: removes customer from this stall
public void setEmpty() {




}
```

7. Type Substitution [10 marks]

Consider the following two classes.

```
class TypeA {
    protected List<Integer> results = new ArrayList<>();
}
class TypeB extends TypeA {
    protected List<Integer> inputs = new ArrayList<>();
}
```

And this variable assignment where we are concerned with whether the assignment violates the *Liskov Substitution Principle (LSP)* as we have described in class.

```
TypeA a = new TypeB();
```

For each line below assume that the classes have only the `add` method listed. In the "**Substitutable?**" column enter either:

- **Y** if `TypeB` can be substituted for `TypeA` without violating LSP or
- **N** if doing so is an LSP violation.

You will receive 2 marks for each correct answer and -1 mark for each incorrect answer.

TypeA	TypeB	Substitutable?

`// REQUIRES i and j are odd`
`// MODIFIES this`
`// EFFECT adds i+j to results`
`void add(int i, int j)`
	`// MODIFIES this`
	`// EFFECT adds i+j to results`
	`@Override`
	`void add(int i, int j)`	

`// REQUIRES i and j are odd`
`// MODIFIES this`
`// EFFECT adds i+j to results`
`void add(int i, int j)`
	`// REQUIRES that i+j is even`
	`// MODIFIES this`
	`// EFFECT adds i+j to results`
	`@Override`
	`void add(int i, int j)`	

`// REQUIRES i and j are odd`
`// MODIFIES this`
`// EFFECT adds i+j to results`
`void add(int i, int j)`
	`// REQUIRES that i is less than j`
	`// MODIFIES this`
	`// EFFECT adds i+j to results`
	`@Override`
	`void add(int i, int j)`	

`// MODIFIES this`
`// EFFECT adds i+j to results`
`void add(int i, int j)`
	`// MODIFIES this`
	`// EFFECT adds i+j to results if results`
	`//  does not already contain i+j`
	`@Override`
	`void add(int i, int j)`	

`// MODIFIES this`
`// EFFECT adds i+j to results`
`void add(int i, int j)`
	`// MODIFIES this`
	`// EFFECT adds i+j to results and`
	`//  adds i and j to inputs`
	`@Override`
	`void add(int i, int j)`	

8. Composite Design Pattern [14 marks]

Consider the following class that stores a list of favourite parking stalls.

```
// A list of favourite parking stalls
class Favourites {
    List<Stall> stalls = new ArrayList<>();

    public void add (Stall stall) {
        stalls.add(stall);
    }

    // EFFECTS: returns true if the favourites list contains an empty stall
    public boolean hasEmptyStall() {
        for (Stall stall : stalls)
            if (stall.isEmpty())
                return true;
        return false;
    }
}
```

In this question you will use the *Composite* design pattern to change the `Favourites` class to conform to the following specification.

```
// A list of favourite parking stalls and/or other Favourites
```

a) Draw a UML class diagram to show how *Composite* can be applied to this solution. Include only types that play a role in the pattern and do not list fields or methods.

b) Give the code for the ***component*** type.

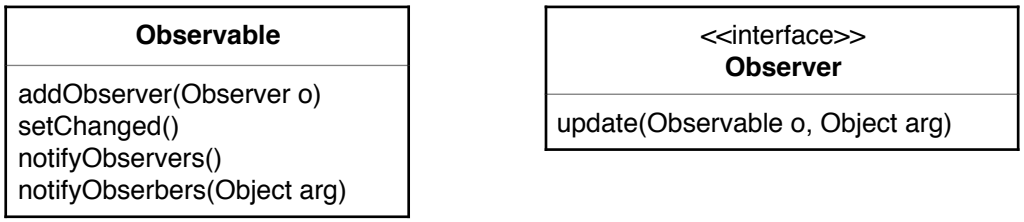
c) Identify which class is the ***leaf*** class and list all changes required to this class. Just the changes.

d) Identify which class is the ***composite*** class and give its complete implementation.

9. Observer Design Pattern [14 marks]

In this question you will use Java's implementation of the *Observer* design pattern (i.e., `Observer` and `Observable`) to modify the code pack so that the `Parkzilla` class calls the `addViolation` method you implemented in Question 5 every time a new violation is added to a `Customer`. Your solution should augment the behaviour of existing classes and not add any new classes.

a) Complete the UML class diagram below to show your design. You must list any method that you added or changed in your concrete subject(s) and observer(s).



b) What changes would you make to the class that is the ***concrete subject*** (i.e., the class being observed)? Include a complete implementation of all modified or added methods, but do not rewrite the entire class.

c) What changes would you make to the class that is the ***concrete observer***? Include a complete implementation of all modified or added methods, but do not rewrite the entire class.

10. Iterator Design Pattern [14 marks]

Note that the `Parkzilla.checkForViolations` method (repeated here for convenience) requires that the `City` class be iterable — but it isn't. In this question you will use Java's implementation of the *Iterator* design pattern to change `City` so this code compiles and executes correctly.

```
public void checkForViolations() {
    for (Stall stall : city) {
        Customer customerInViolation = stall.isInViolation();
        if (customerInViolation!=null) {
            customerInViolation.addViolation();
        }
    }
}
```

Note that `City` has a map containing multiple `Lot` objects and each `Lot` contains a collection of `Stall` objects. Iterating over the `City` requires iterating over every stall in every parking lot in the city. You will do this in two steps.

a) List all of the modifications required to make the `Lot` class iterable so that, for example, the following code would compile and execute correctly.

```
int countEmptyStalls(Lot lot) {
    int count = 0;
    for (Stall stall : lot) {
        if (stall.isEmpty())
            count += 1;
    }
    return count;
}
```

Do this in the simplest possible way; e.g., you **can** call the Java `iterator` method in your solution. Give a complete implementation of all modified or added methods, but do not rewrite the entire class.