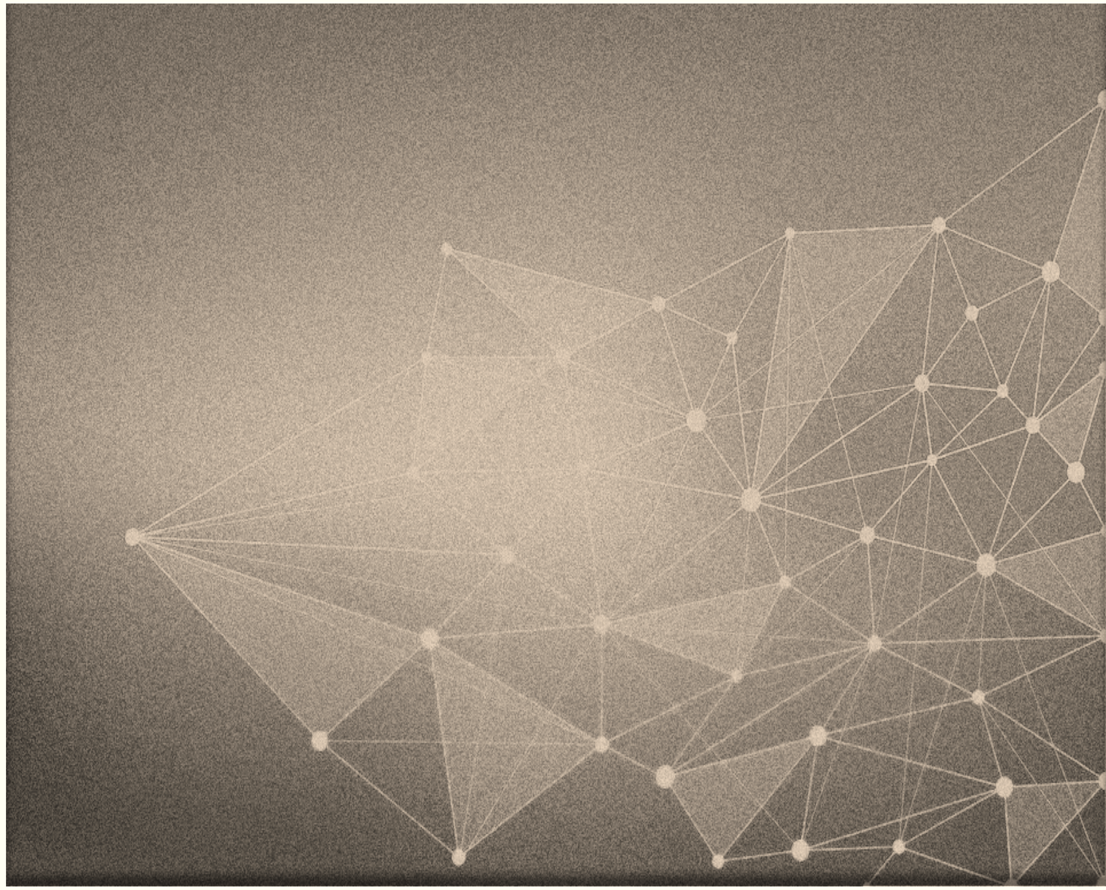
EECS2030: ADVANCED OBJECT-ORIENTED PROGRAMMING

By: Dr. Marzieh Ahmadzadeh



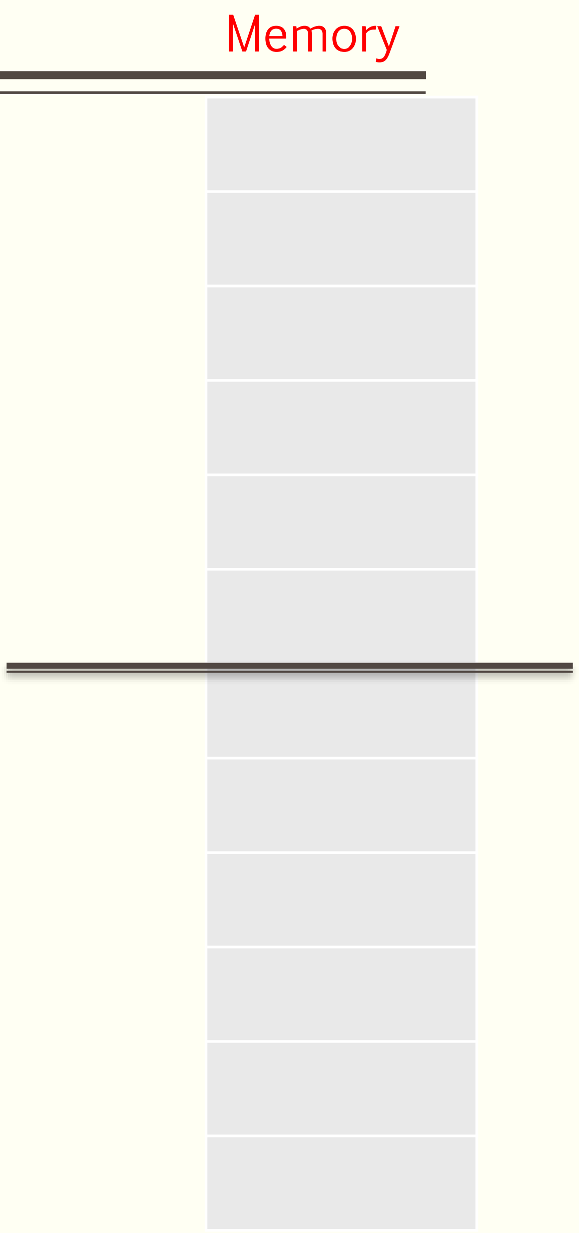
Outline

- Last Week:
  - Static & Non-static variables and Methods
  - Design By Contract
    - Implications for unit testing
    - JUnit to test exception
- This Week
  - Different kinds of copying an object reference
    - Aliasing, Shallow and Deep copying
  - Objects relationship
    - Aggregation
    - Composition
  - Privacy Leak

ALIASING

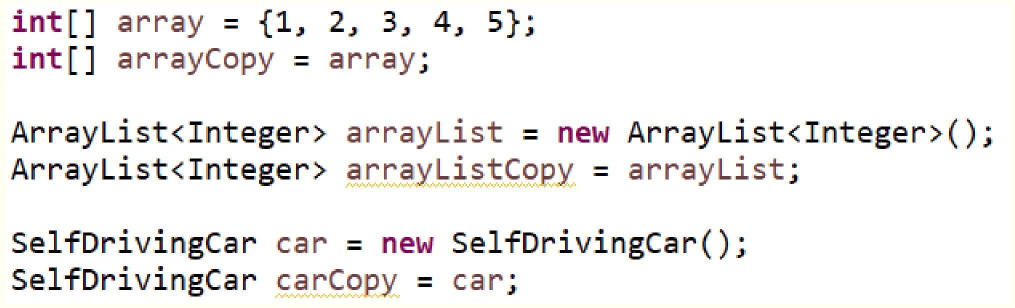
Aliasing

- Aliasing is the creation of a second copy of a `reference` variable using `assignment` operator.



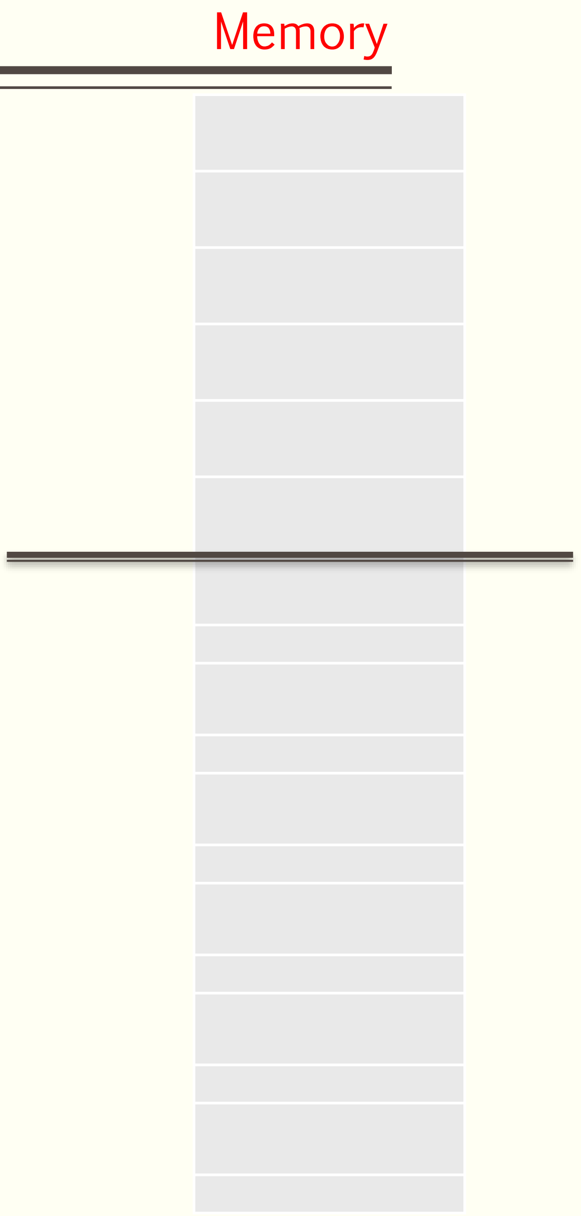
Aliasing

- Aliasing is the creation of a second copy of a `reference` variable using `assignment` operator.



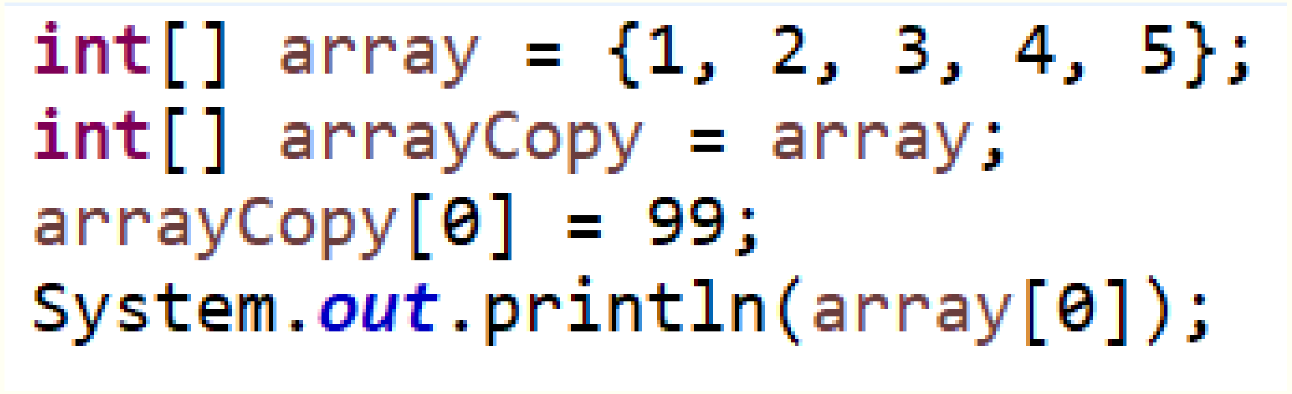
```java
int[] array = {1, 2, 3, 4, 5};
int[] arrayCopy = array;

ArrayList<Integer> arrayList = new ArrayList<Integer>();
ArrayList<Integer> arrayListCopy = arrayList;

SelfDrivingCar car = new SelfDrivingCar();
SelfDrivingCar carCopy = car;
```

- Both the references point to the same object, so they are `aliases`.

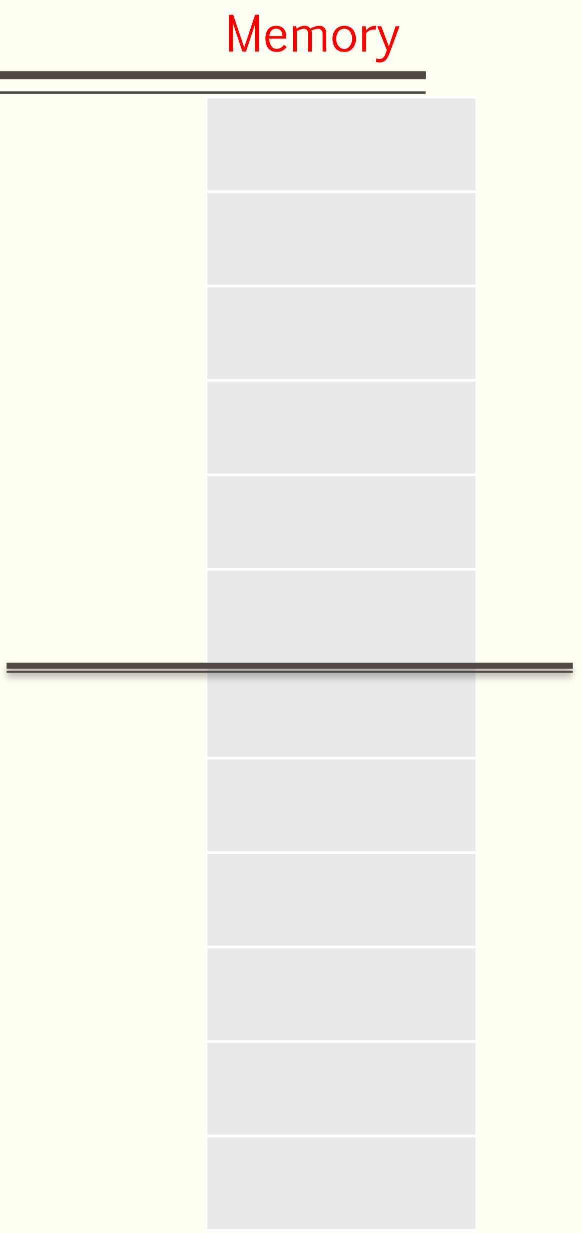


How Does Aliasing work?

- Creating an alias is like cutting a `duplicate key` for a house.
- What is the output of this code?

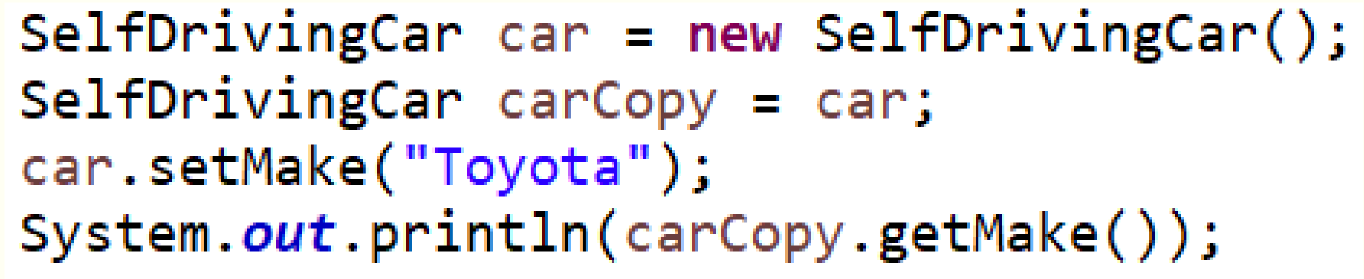


```java
int[] array = {1, 2, 3, 4, 5};
int[] arrayCopy = array;
arrayCopy[0] = 99;
System.out.println(array[0]);
```

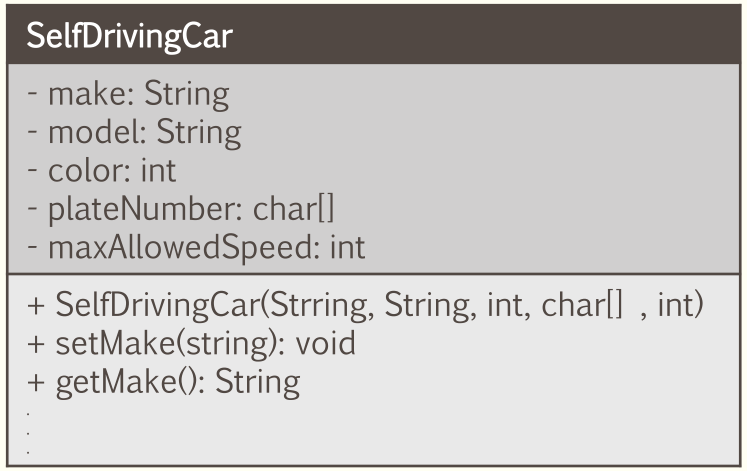


How Does Aliasing work?

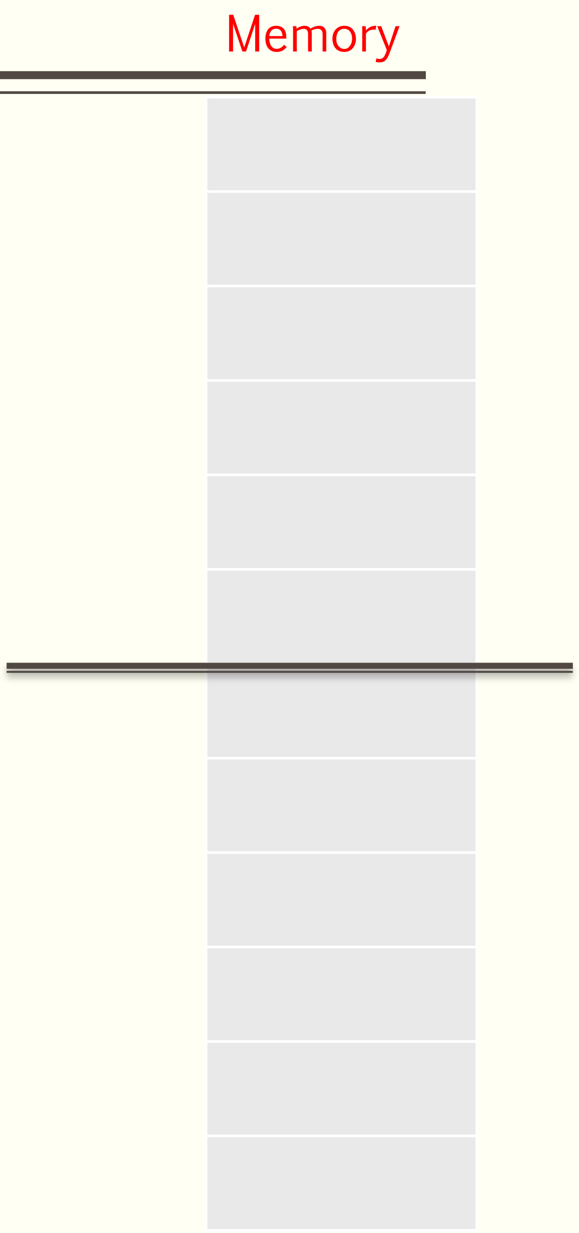
- Creating an alias is like cutting a `duplicate key` for a house.
- What is the output of this code?



```
SelfDrivingCar car = new SelfDrivingCar();
SelfDrivingCar carCopy = car;
car.setMake("Toyota");
System.out.println(carCopy.getMake());
```

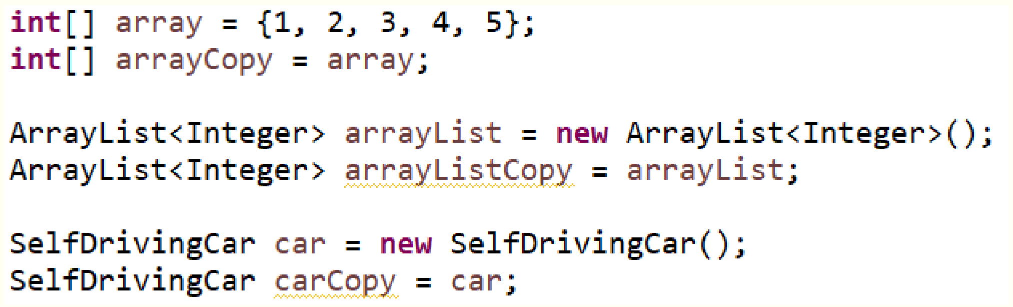


```
SelfDrivingCar
- make: String
- model: String
- color: int
- plateNumber: char[]
- maxAllowedSpeed: int
+ SelfDrivingCar(Strring, String, int, char[] , int)
+ setMake(string): void
+ getMake(): String
.
.
.
```

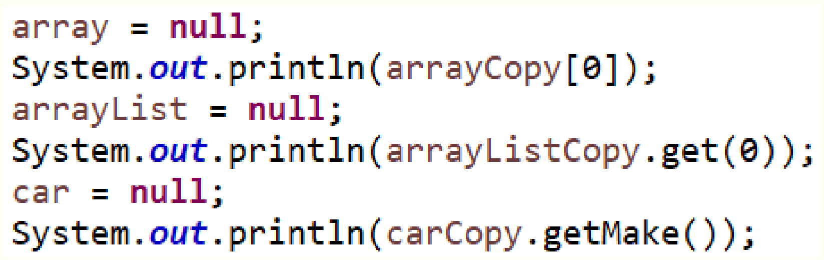


Aliasing Side Effect

- What if the object is no longer required?

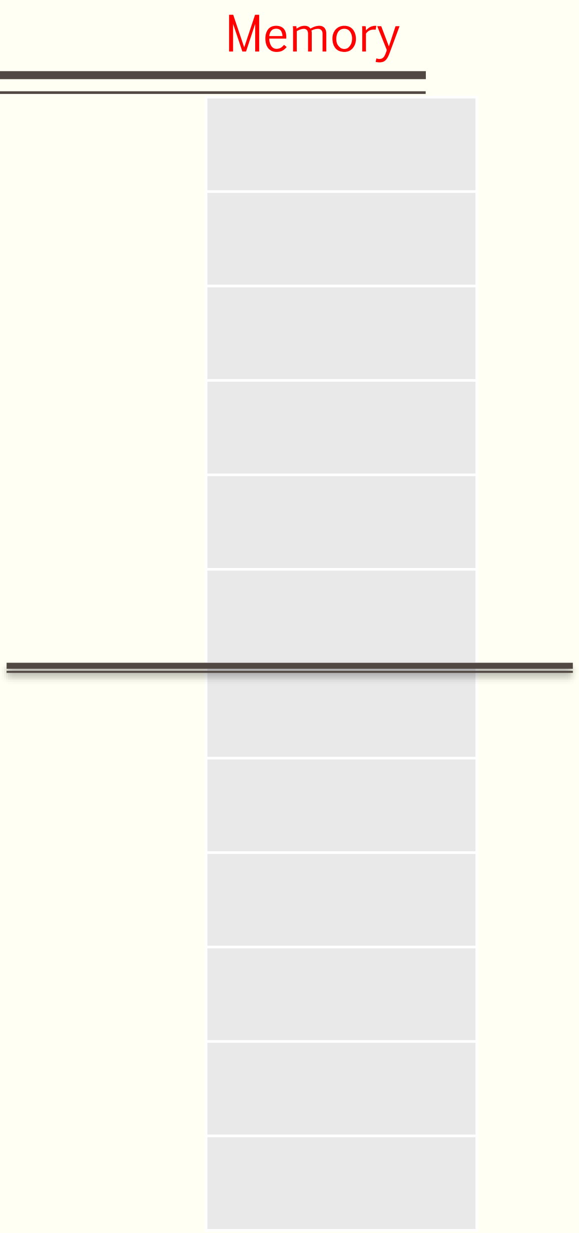


```
int[] array = {1, 2, 3, 4, 5};
int[] arrayCopy = array;

ArrayList<Integer> arrayList = new ArrayList<Integer>();
ArrayList<Integer> arrayListCopy = arrayList;

SelfDrivingCar car = new SelfDrivingCar();
SelfDrivingCar carCopy = car;
```



```
array = null;
System.out.println(arrayCopy[0]);
arrayList = null;
System.out.println(arrayListCopy.get(0));
car = null;
System.out.println(carCopy.getMake());
```

- Don't forget to nullify all the aliases, if the object is no longer required, otherwise GC cannot perform its job.



DEEP COPYING

Deep Copying

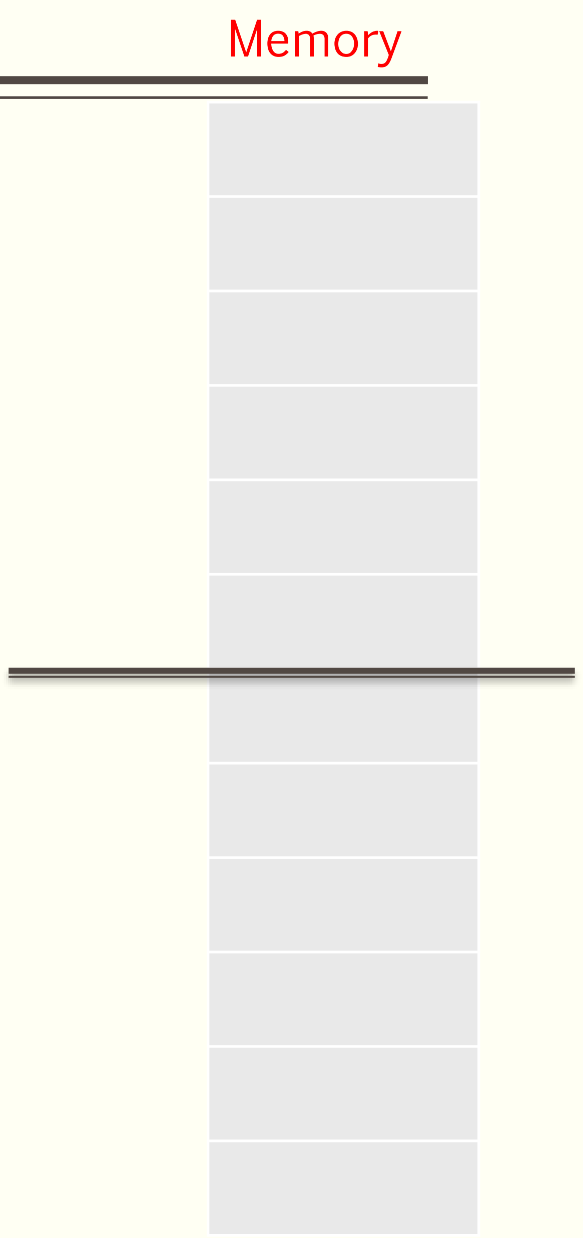
- Aliasing is not a solution, when you need a `clone` of your object.
- Cloning refers to create an exact same object but in a different memory space.
  - The key is to request for a new space using `new` operator.
- By deep copying you can create a clone of an object.
- So changing one object does not affect the other.

Deep Copying

- How to deep copy an array?

```java
char [] charArray = {'A', 'B', 'C'};
```

```java
char [] charArrayClone = new char [charArray.length];
for (int i = 0; i < charArray.length; i++)
    charArrayClone[i] = charArray[i];
```

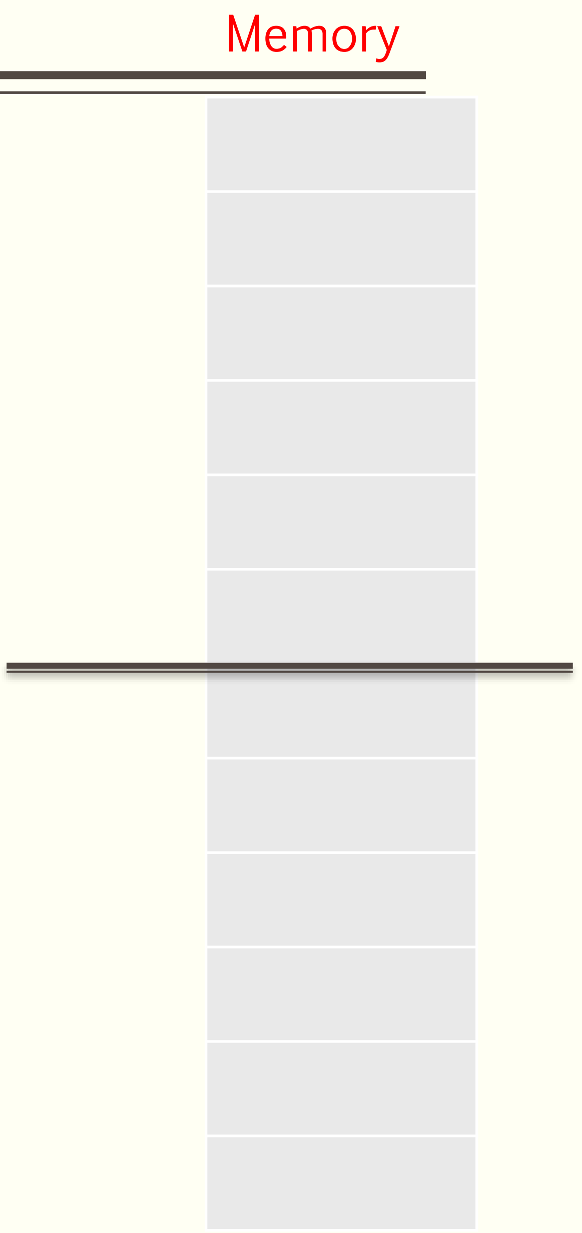


Deep Copying

- How to deep copy an arrayList?

```java
ArrayList<Double> grade = new ArrayList<Double>();
grade.add(78.9);
grade.add(89.7);
```

```java
ArrayList<Double> gradeClone = new ArrayList<Double>();
    for (int i = 0; i < grade.size(); i++)
        gradeClone.add(grade.get(i));
```



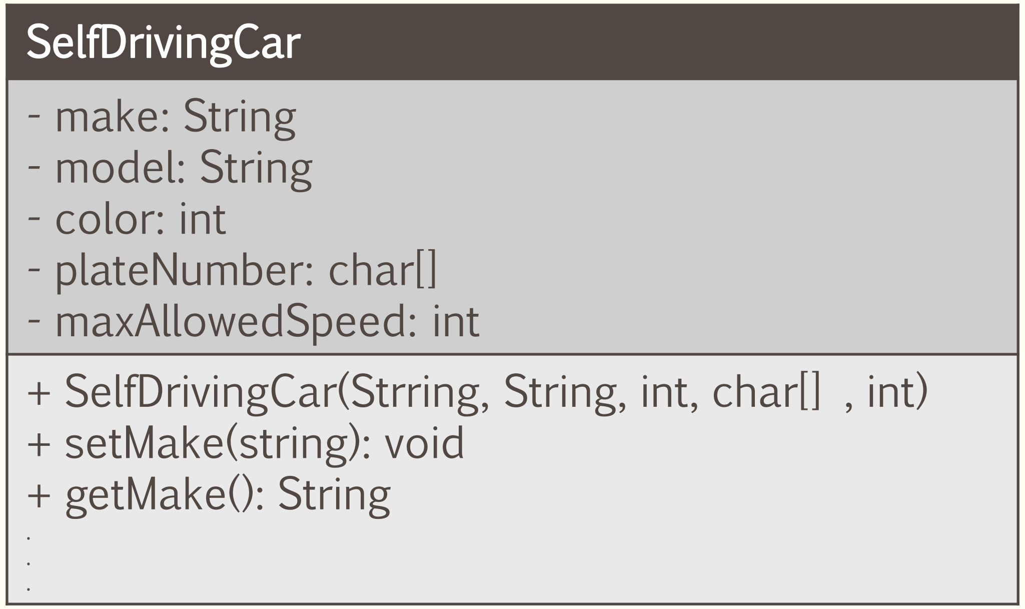
Deep Copying

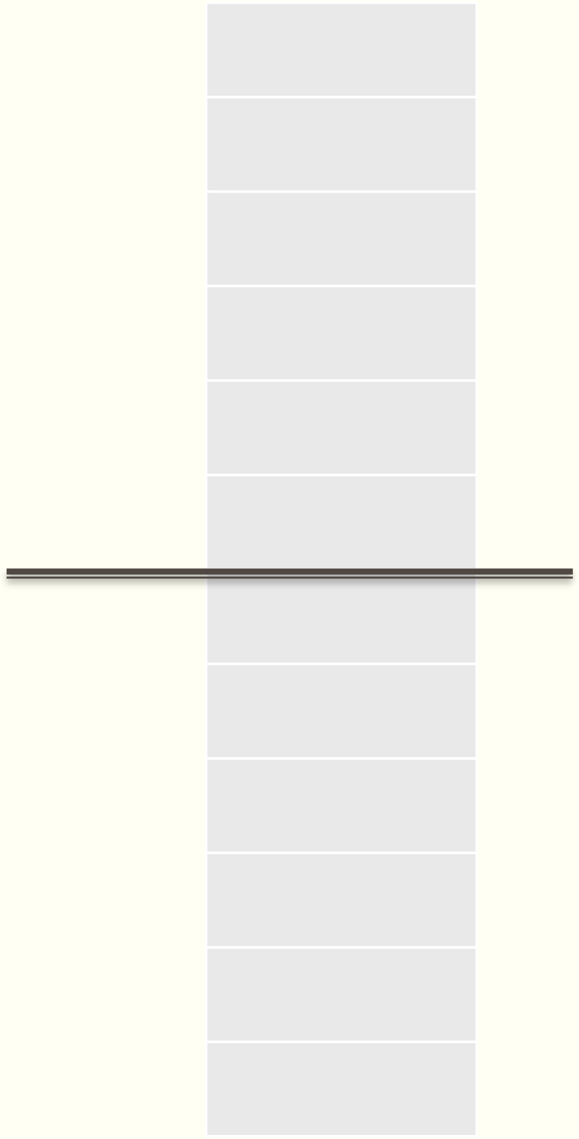
Memory

- How to deep copy a car?

```
String plate = "xxx000";
char [] plateNumber = plate.toCharArray();
SelfDrivingCar driverlessCar = new SelfDrivingCar("Toyota", "Rav4", 220355235, plateNumber ,110);
```

```
SelfDrivingCar carClone = new SelfDrivingCar(driverlessCar.getMake(),
                                              driverlessCar.getModel(),
                                              driverlessCar.getColor(),
                                              driverlessCar.getPlateNumber(),
                                              driverlessCar.getMaxAllowedSpeed());
```

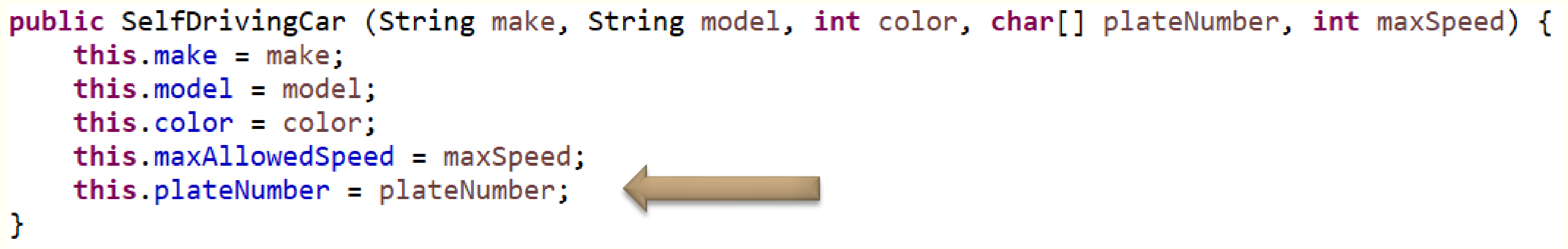




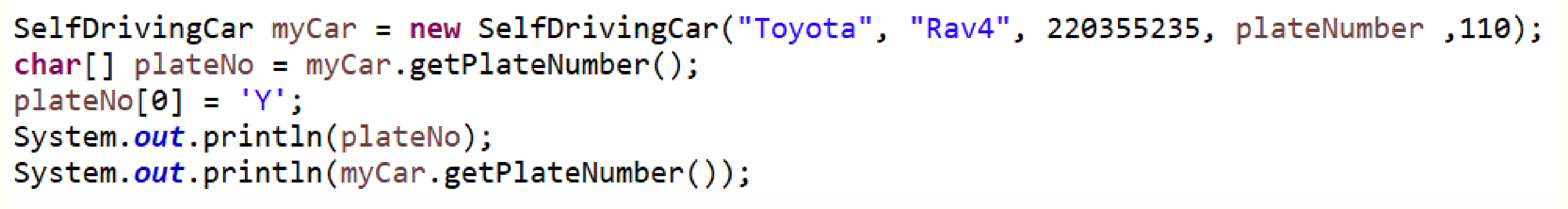
Question:

Memory

- How should the constructor for SelfDrivingCar look like?



```java
public SelfDrivingCar (String make, String model, int color, char[] plateNumber, int maxSpeed) {
    this.make = make;
    this.model = model;
    this.color = color;
    this.maxAllowedSpeed = maxSpeed;
    this.plateNumber = plateNumber;
}
```

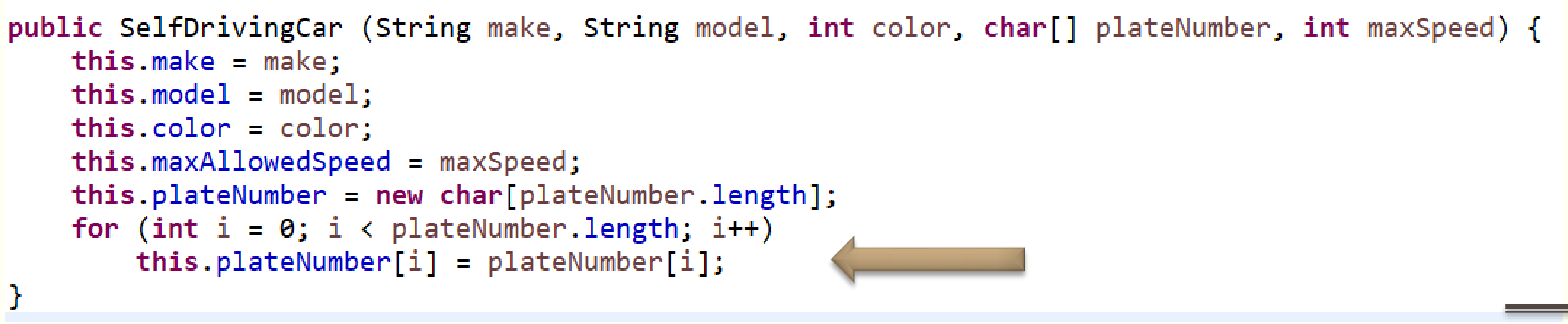


```java
SelfDrivingCar myCar = new SelfDrivingCar("Toyota", "Rav4", 220355235, plateNumber ,110);
char[] plateNo = myCar.getPlateNumber();
plateNo[0] = 'Y';
System.out.println(plateNo);
System.out.println(myCar.getPlateNumber());
```

This is NOT a deep copy!

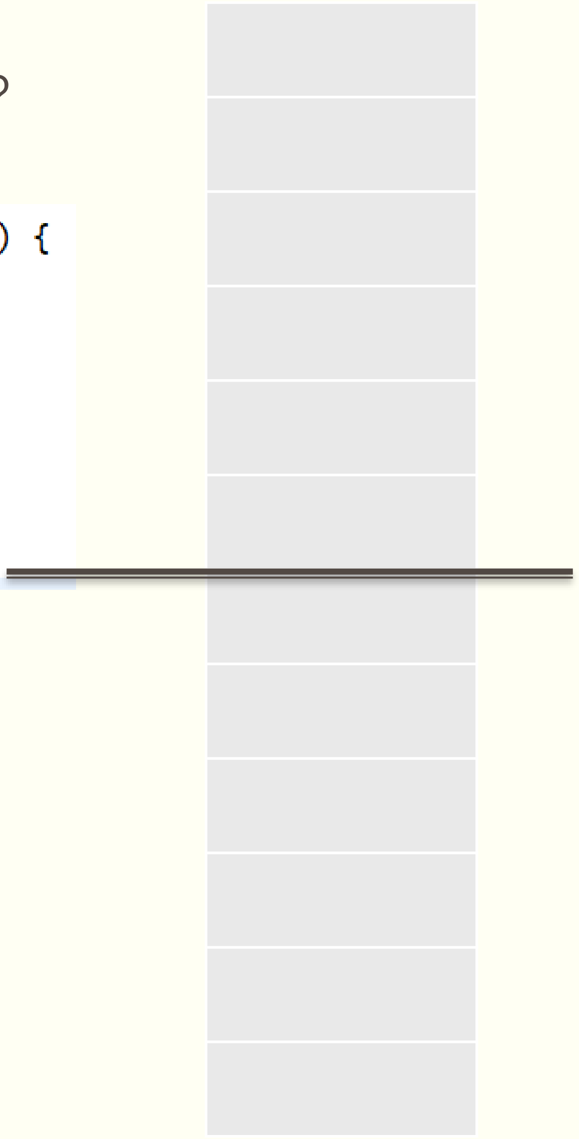
Question:

- How should the constructor for SelfDrivingCar look like with deep copying?



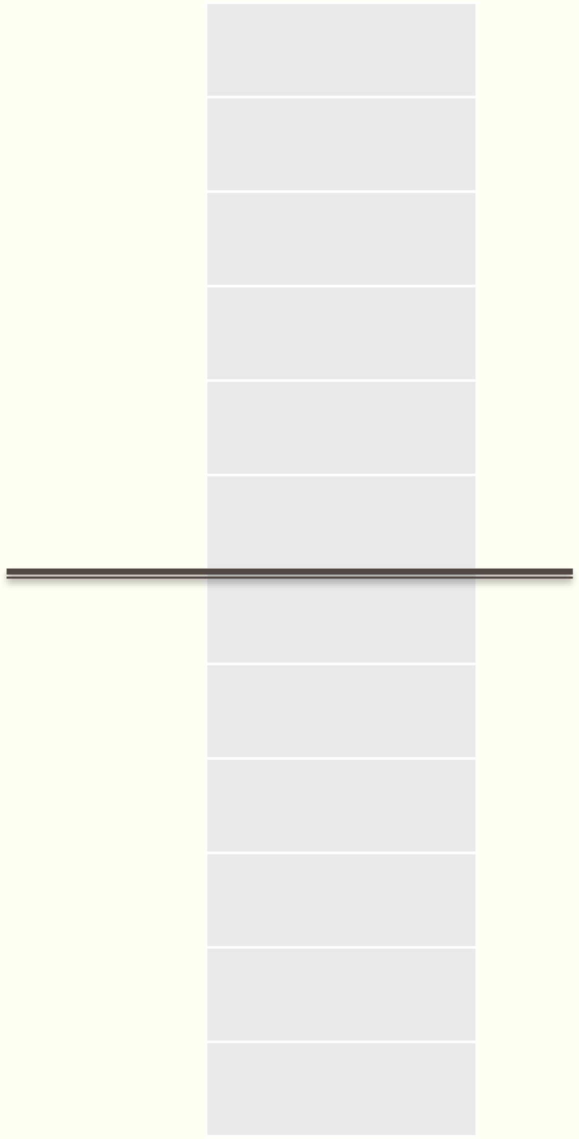
```
public SelfDrivingCar (String make, String model, int color, char[] plateNumber, int maxSpeed) {
    this.make = make;
    this.model = model;
    this.color = color;
    this.maxAllowedSpeed = maxSpeed;
    this.plateNumber = new char[plateNumber.length];
    for (int i = 0; i < plateNumber.length; i++)
        this.plateNumber[i] = plateNumber[i];
}
```

Memory

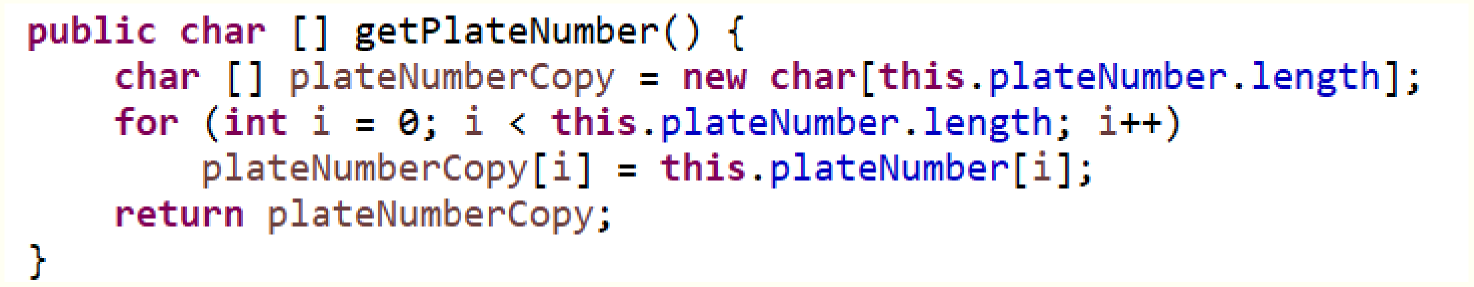


Question:

Memory



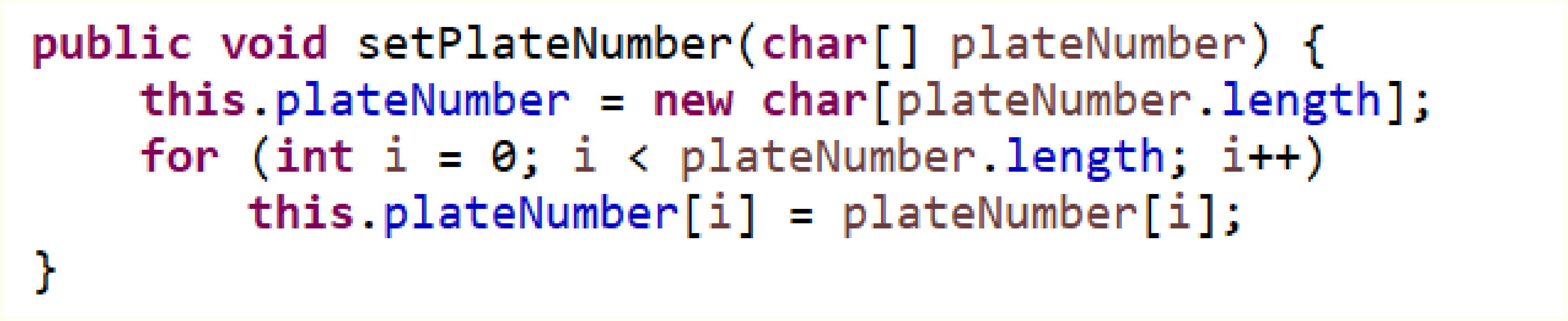
- How should the accessors for SelfDrivingCar's attribute look like with deep copying?



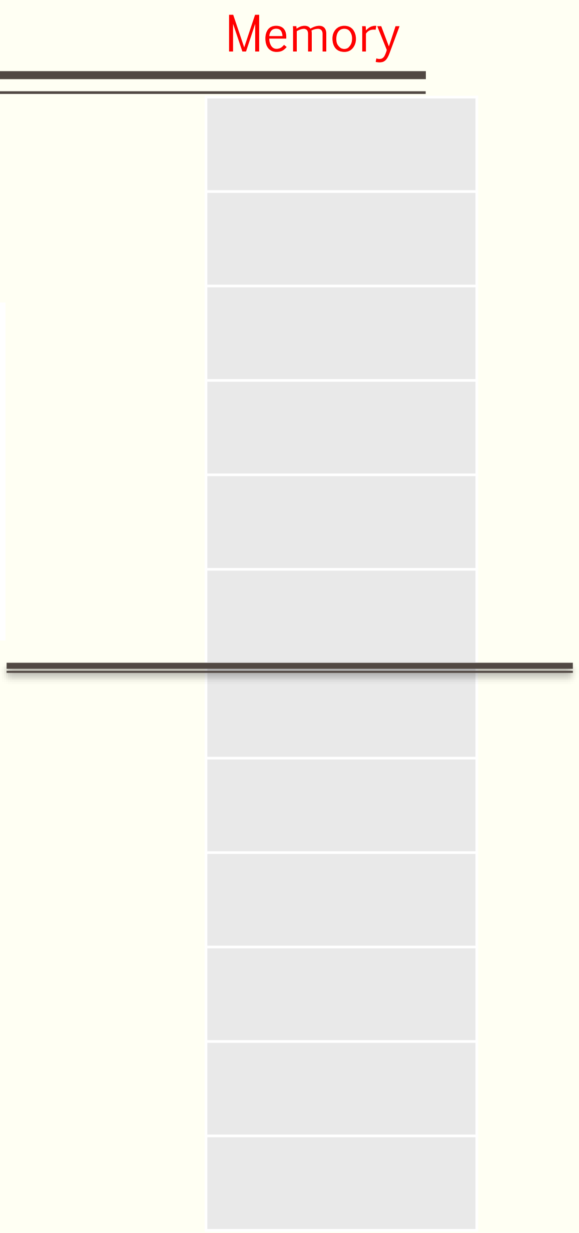
```
public char [] getPlateNumber() {
    char [] plateNumberCopy = new char[this.plateNumber.length];
    for (int i = 0; i < this.plateNumber.length; i++)
        plateNumberCopy[i] = this.plateNumber[i];
    return plateNumberCopy;
}
```

Question:

- How should the mutators for SelfDrivingCar's attributes look like with deep copying?

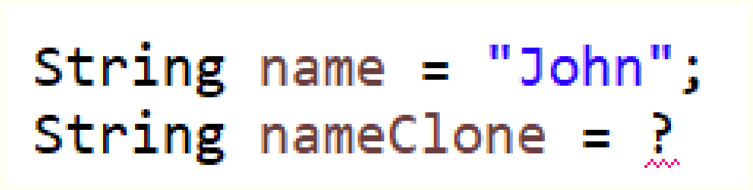


```java
public void setPlateNumber(char[] plateNumber) {
    this.plateNumber = new char[plateNumber.length];
    for (int i = 0; i < plateNumber.length; i++)
        this.plateNumber[i] = plateNumber[i];
}
```

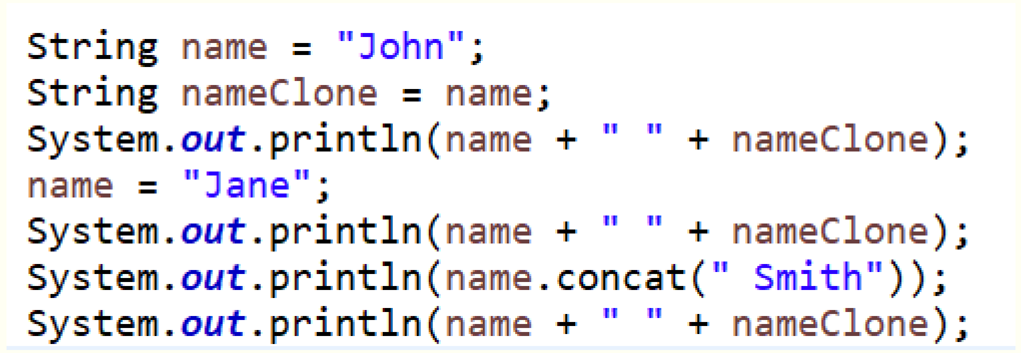


Deep Copying

- How do I deep copy a string?

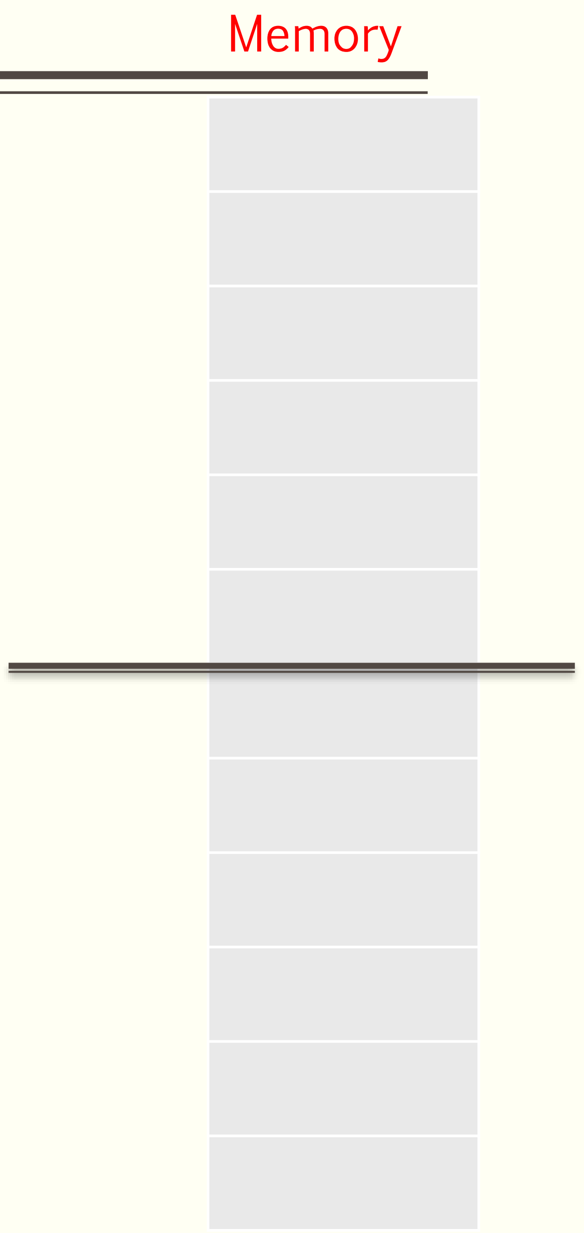


```
String name = "John";
String nameClone = ?
```



```
String name = "John";
String nameClone = name;
System.out.println(name + " " + nameClone);
name = "Jane";
System.out.println(name + " " + nameClone);
System.out.println(name.concat(" Smith"));
System.out.println(name + " " + nameClone);
```

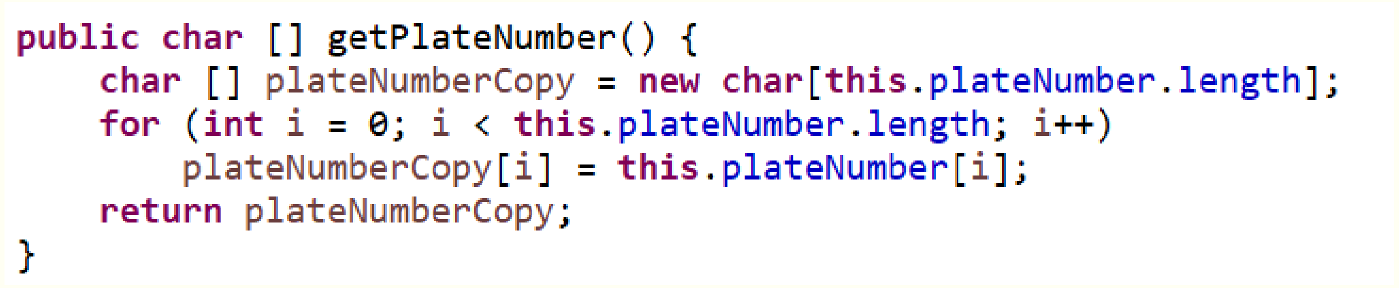
String class developers have been careful about coping. All the copying in the String methods are deep. With deep copying they preserved the immutability feature of the object.



IMMUTABILITY (REVIEW)

Recall: Immutability

- If the state of an object cannot change after it is constructed, it is said that the object is immutable.
- You can create an immutable object if you do not let any method changes the state of the object.
- This requires you to
  a) get rid of all mutator methods.
  b) deep copy the reference variables in the accessor methods.
     - This means no accessor method should return a reference to an instance variable.



```java
public char [] getPlateNumber() {
    char [] plateNumberCopy = new char[this.plateNumber.length];
    for (int i = 0; i < this.plateNumber.length; i++)
        plateNumberCopy[i] = this.plateNumber[i];
    return plateNumberCopy;
}
```

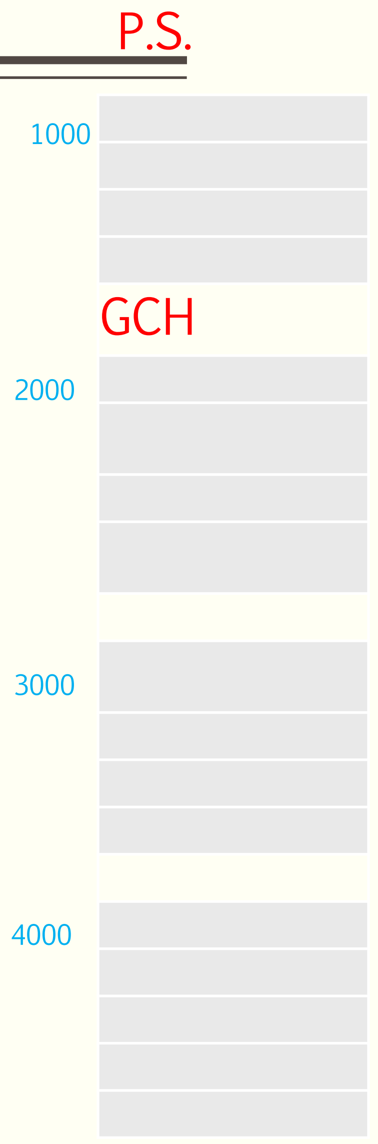
SHALLOW COPYING

Shallow Copying

- Shallow copy is as useful as deep copy.
- It clones the component of the objects but not the sub-components.
  - It creates an alias of the subcomponent.

```java
public class Account {
    char accountType;
    int accountNumber;
    double balance;
    Date dateOpened;

    public Account() {
        accountType = ' ';
        accountNumber = 0;
        balance = 0;
        dateOpened = new Date();
    }

    public Account (char accType, int accNumber, double balance, Date openedDate) {
        this.accountType = accType;
        this.accountNumber = accNumber;
        this.balance = balance;
        this.dateOpened = openedDate;
    }

    public Account (Account acc) {
        this (acc.accountType, acc.accountNumber, acc.balance, acc.dateOpened);
    }
}
```

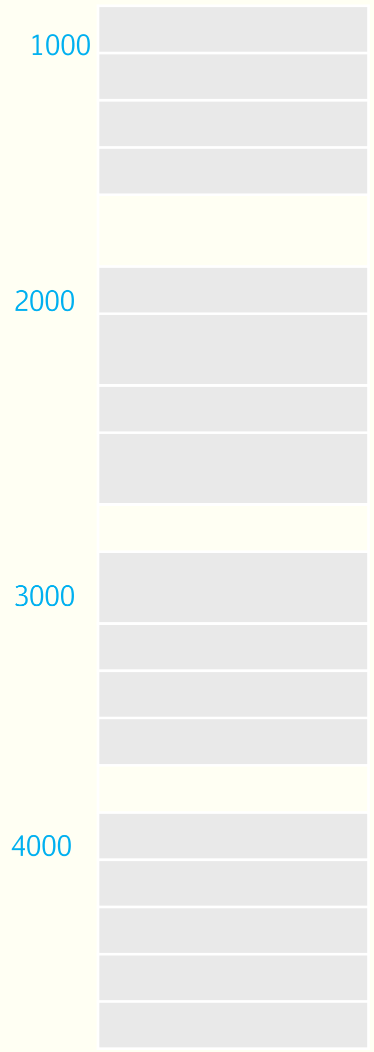
```java
Account firstAcc = new Account('C', 100200, 100, new Date());
Account secondAcc = new Account(firstAcc);
```



Shallow Coping: Example

GCH

```
import java.util.Calendar;
import java.util.ArrayList;

public class Customer {
    String name;
    Calendar dob;
    ArrayList<Account> account;

    public Customer(String custName, Calendar dofb, ArrayList<Account> acc) {
        //deep copy
        this.name = new String (custName);
         // use of static factory method to create an object of Calendar
        this.dob = Calendar.getInstance();
        // deep copy
        this.dob.set(dofb.get(Calendar.YEAR), dofb.get(Calendar.MONTH), dofb.get(Calendar.DAY_OF_MONTH));
        //shallow copy
        this.account = new ArrayList<Account>();
        for (int i = 0; i < acc.size(); i++)
            this.account.add(acc.get(i));
    }
}
```



- In-class Activity
- Break

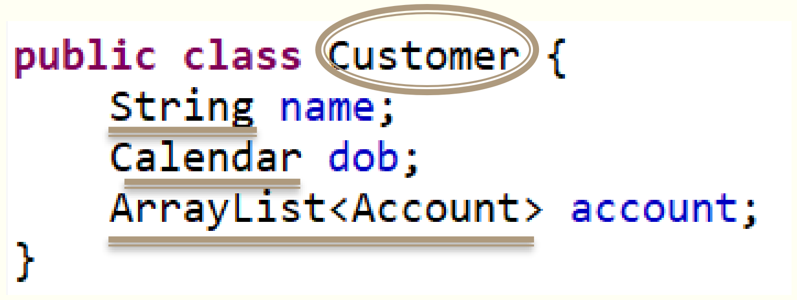
OBJECTS RELATIONSHIPS

Object Relationships

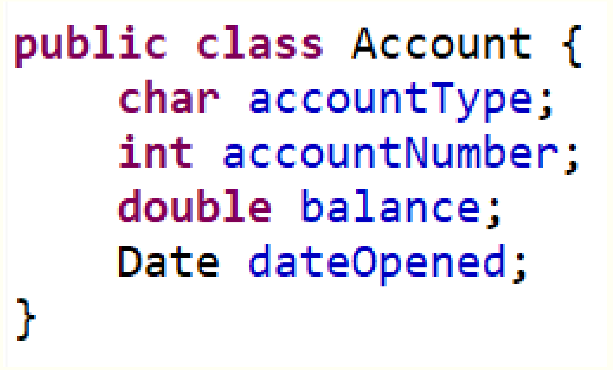
- In object-oriented programming, objects have relationships with each other.
  - This makes code-reuse possible.
- Object relationship: When (the properties of) an object (i.e. its instance variable and methods) are used in another object, so you don't have to write the code again.
- Types of relationships:
  - Aggregation: Has-a relationship
  - Composition: Has-a relationship
  - Inheritance: will be discussed next lecture. Is-A relationship.

Has-A relationship

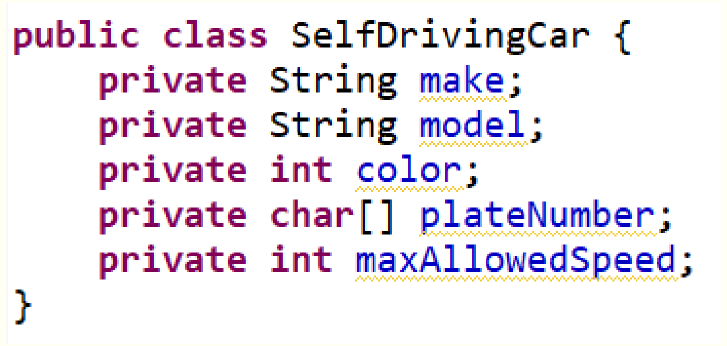
- This relationship describes a situation when an object has an instance variable, whose type is non-primitive.



```
public class Customer {
    String name;
    Calendar dob;
    ArrayList<Account> account;
}
```



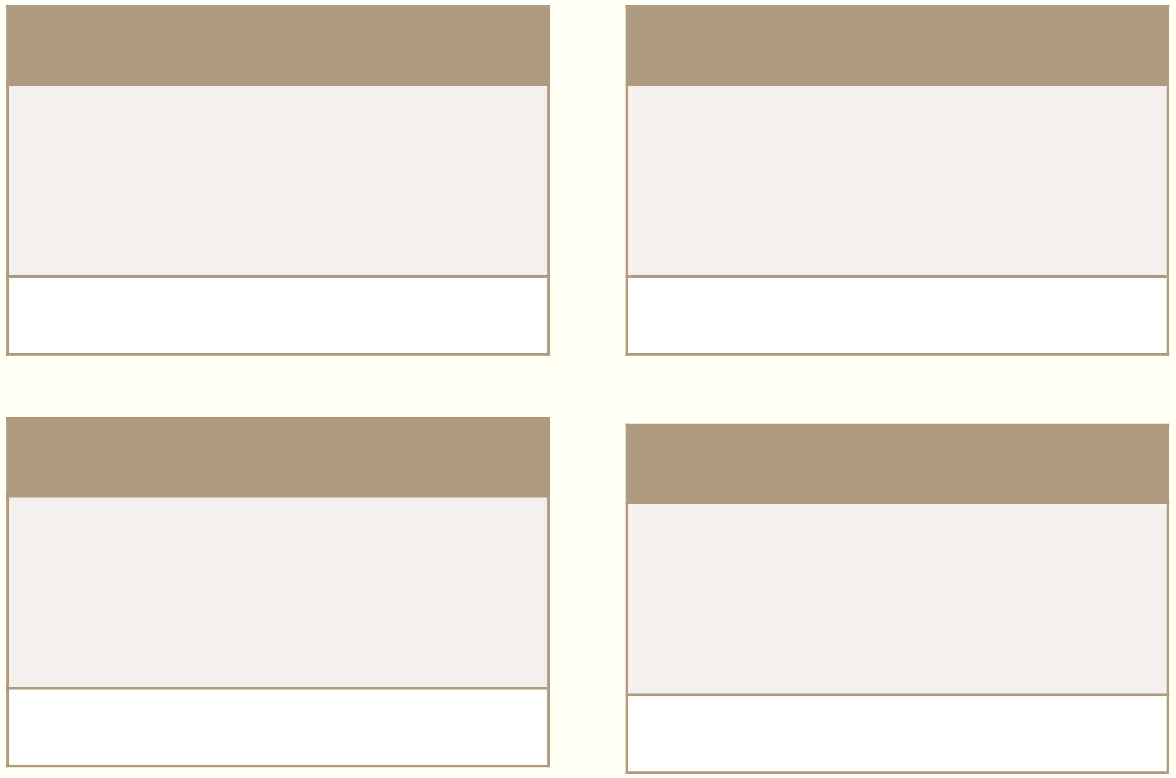
```
public class Account {
    char accountType;
    int accountNumber;
    double balance;
    Date dateOpened;
}
```



```
public class SelfDrivingCar {
    private String make;
    private String model;
    private int color;
    private char[] plateNumber;
    private int maxAllowedSpeed;
}
```

Example: Has- A relationship

- A student has-a couple of courses.
- A house has-a couple of rooms.
- A person has-a couple of accounts.
- A person has-a date of birth.
- A car has-a collection of parts.
- A family physician has-a list of patients.
- A shop has-a collection of products.
- Later in Data Structure course:
    - A linked list has-a set of nodes.
    - A tree has-a set of nodes.



The types of Has-A relationships

- Two types :
  - Aggregation: Weak Association. The components can exist independent of the object
  - Composition: Strong Association. The components cannot exist independent of the object as the object own them.

- Difference in term of programming
  - Aggregation: uses aliases or shallow copy to create the object.
  - Composition: uses deep copy to create the object.

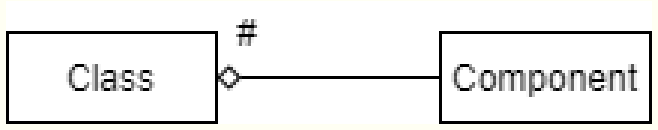
AGGREGATION

Aggregation

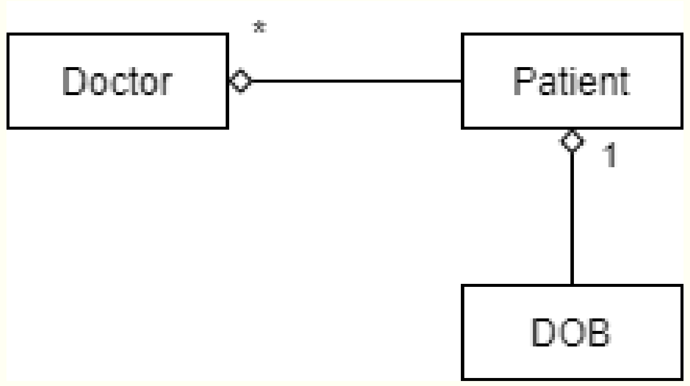
- The relationship between objects is weak.
- A student has-a couple of courses.
  - What if a student graduates?
- A family physician has-a list of patients.
  - What happened if the family physician is retired?
- With aggregation the object does not own its component. So the component can exist independent of the object.
- If the object is cleaned up from memory, the component can still be there.

UML

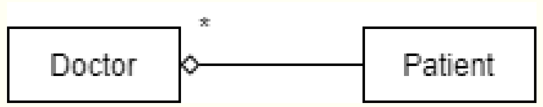
The object HAS # number of component object



Shows the multiplicity of the component.



Aggregation: Example; Constructors



```java
public class Patient {
    String name;
    Calendar dob;

    public Patient(String name, Calendar dob) {
        this.name = name;
        this.dob = dob;
    }
}
```

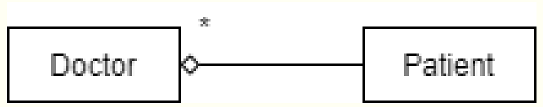
```java
public class Doctor {
    private String name;
    private ArrayList<Patient> patient;
}
```

```java
public Doctor(String name, ArrayList<Patient> patient) {
    this.name = name;
    this.patient = new ArrayList<Patient>();
    for(int i = 0; i < patient.size(); i++)
        this.patient.add(patient.get(i));

}
```

```java
public Doctor (Doctor doctor) {
    this.name = doctor.name;
    this.patient = doctor.patient;
}
```

Note: Aliasing / Shallow copy

Aggregation: Example; Mutators



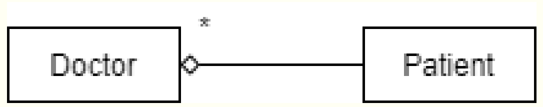
```
public class Patient {
    String name;
    Calendar dob;

    public Patient(String name, Calendar dob) {
        this.name = name;
        this.dob = dob;
    }
}
```

```
public class Doctor {
    private String name;
    private ArrayList<Patient> patient;
}
```

```
public void setName(String name) {
    this.name = name;
}
public void setPatient(ArrayList<Patient> patient) {
    this.patient = new ArrayList<Patient>();
    for(int i = 0; i < patient.size(); i++)
        this.patient.add(patient.get(i));
}
```

Note: Aliasing / Shallow copy

Aggregation: Example; Accessors



```
public class Patient {
    String name;
    Calendar dob;

    public Patient(String name, Calendar dob) {
        this.name = name;
        this.dob = dob;
    }
}
```

```
public class Doctor {
    private String name;
    private ArrayList<Patient> patient;
}
```

```
public String getName() {
    return this.name;
}
public ArrayList<Patient> getPatient() {
    return this.patient;
}
```

Note: Aliasing / Shallow copy

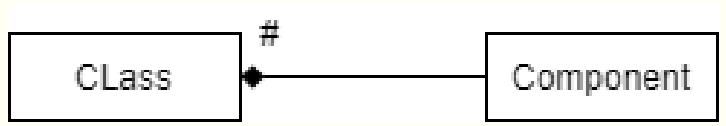
COMPOSITION

Composition

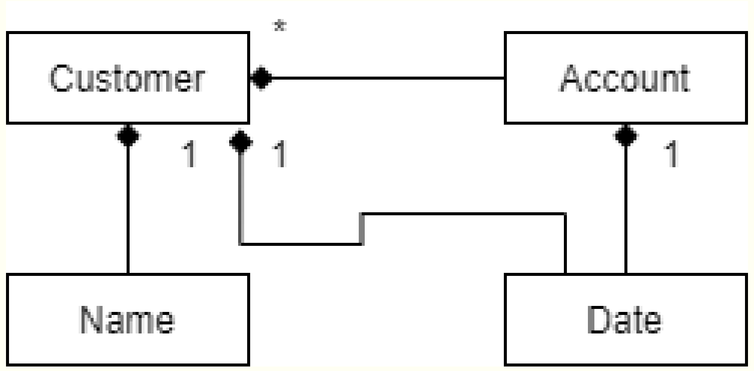
- The relationship between objects is strong.
  - The object owns its components.
  - The object has exclusive access to it component.
  - The components cannot live independent of the object.
  - If the object is cleaned up from memory, all its component will be gone too.
- A house has-a couple of rooms.
  - Do the rooms exists if the house is demolished?
- A customer has-a couple of accounts.
  - Can the accounts be used for other customers if a person decides not to be a customer anymore?
- A shop has-a collection of products.
  - Can the product exist if the shop is closed down?

UML

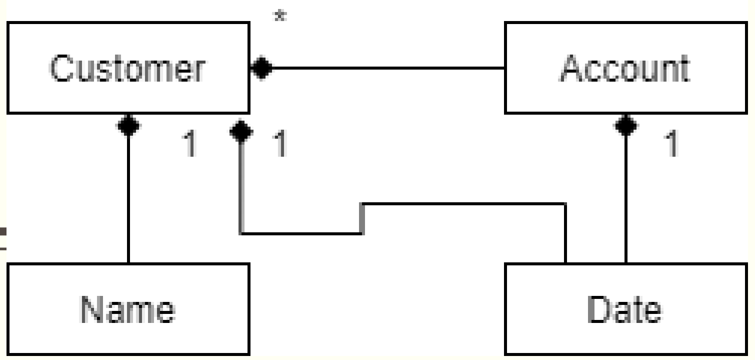
The object HAS # number of component object



Shows the multiplicity of the component.



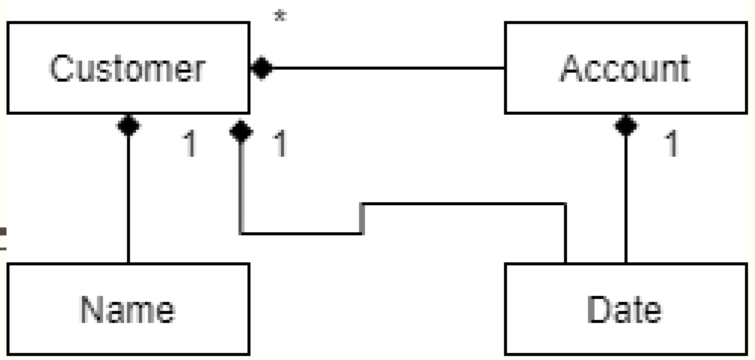
Composition: Example; Constructors



```
public class Account {
    char accountType;
    int accountNumber;
    double balance;
    Date dateOpened;
}
```

```
public class Customer {
    String name;
    Calendar dob;
    ArrayList<Account> account;

    public Customer(String custName, Calendar dofb, ArrayList<Account> acc) {
        this.name = new String (custName);
        this.dob = Calendar.getInstance();
        this.dob.set(dofb.get(Calendar.YEAR), dofb.get(Calendar.MONTH), dofb.get(Calendar.DAY_OF_MONTH));
        this.account = new ArrayList<Account>();
        for (int i = 0; i < acc.size(); i++)
            this.account.add(new Account(acc.get(i)));
    }
}
```

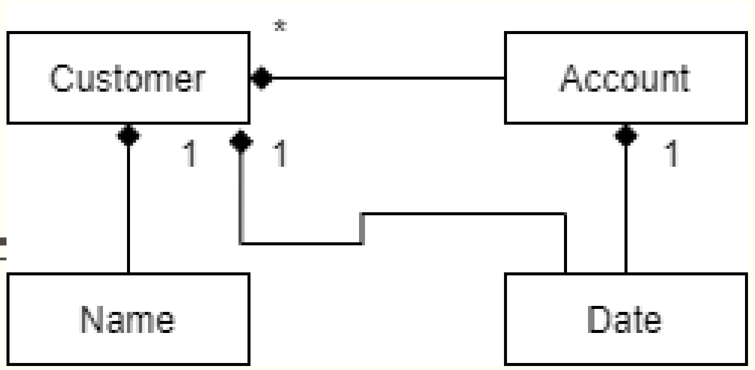
Composition: Example; Mutators



```
public class Account {
    char accountType;
    int accountNumber;
    double balance;
    Date dateOpened;
}
```

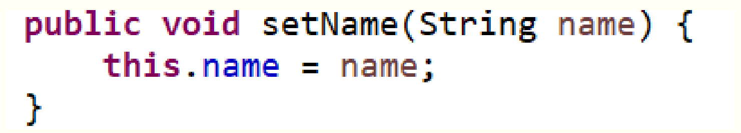
```
public void setName(String name) {
    this.name = name;
}
public void setDob(Calendar dofb) {
    this.dob = Calendar.getInstance();
    this.dob.set(dofb.get(Calendar.YEAR), dofb.get(Calendar.MONTH), dofb.get(Calendar.DAY_OF_MONTH));
}
public void setAccount(ArrayList<Account> acc){
    this.account = new ArrayList<Account>();
    for (int i = 0; i < acc.size(); i++)
        this.account.add(new Account(acc.get(i)));

}
```

Composition: Example; Accessors



```
public class Account {
    char accountType;
    int accountNumber;
    double balance;
    Date dateOpened;
}
```

```
public String getName() {
    return this.name;
}
public Calendar getDob() {
    Calendar db = Calendar.getInstance();
    db.set(this.dob.get(Calendar.YEAR), this.dob.get(Calendar.MONTH), this.dob.get(Calendar.DAY_OF_MONTH));
    return db;
}
public ArrayList<Account> getAccount(){
    ArrayList<Account> acc = new ArrayList<Account>();
    for (int i = 0; i < this.account.size(); i++)
        acc.add(new Account(this.account.get(i)));
    return acc;

}
```



```
public void setName(String name) {
    this.name = name;
}
```

Question

- While this example is all about composition, why didn't I have to create a new object for the name, before I initialize it or assign a new value to it?
- Answer: String is an immutable object.
- Immutability is a useful tool to create composition.
- In this example, if ArrayList, Account and Calendar were immutable, then I could have treated them the same way that I treated the name.

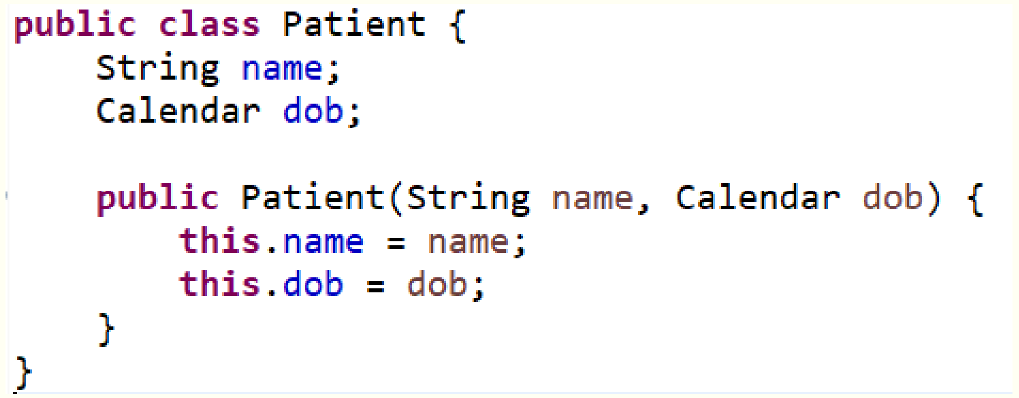
PRIVACY LEAK

Privacy Leak

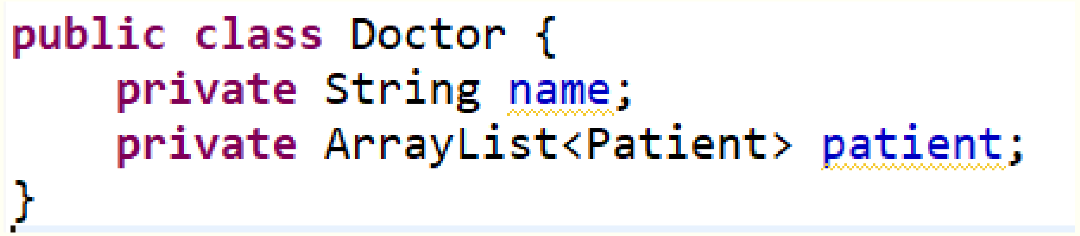
- Privacy leak is a situation where a client get access to the data that they should not get access to it.
- Privacy leak happens when a class exposes a reference to an attribute, which was not supposed to be public.
- This only applies to non-primitive attribute.
    - Primitive and immutable attributes are privacy leak resistance.

Privacy Leak: Example

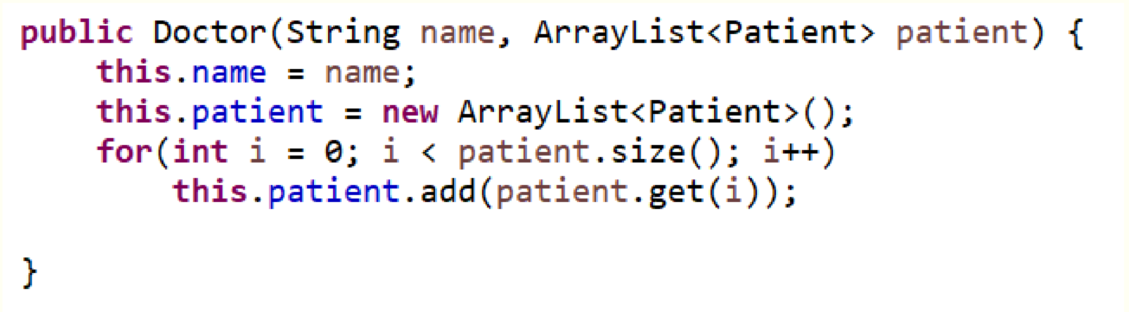
- Assume that a doctor was supposed to be a composition of the patients.
- Privacy leak can be seen in the constructor.



```java
public class Patient {
    String name;
    Calendar dob;

    public Patient(String name, Calendar dob) {
        this.name = name;
        this.dob = dob;
    }
}
```



```java
public class Doctor {
    private String name;
    private ArrayList<Patient> patient;
}
```



```java
public Doctor(String name, ArrayList<Patient> patient) {
    this.name = name;
    this.patient = new ArrayList<Patient>();
    for(int i = 0; i < patient.size(); i++)
        this.patient.add(patient.get(i));

}
```

Expectations & Reading

- Expectations:
  - You should be able to explain what aliasing is.
  - You must have a thorough understanding of different types of copying.
  - You should be able to present the copying process and aliasing in a memory diagram.
  - You should be able to implement an immutable class if requested.
  - You must know the difference between the properties of aggregation and composition relationship.
  - You should be able to decide if the association between two objects is aggregation or composition [depending on the description of the problem].
  - You should be able to identify the situation in which privacy leak happens.
- Reading
  - Nothing really is better than these slides for this topic, I believe ☺
- One-minute paper