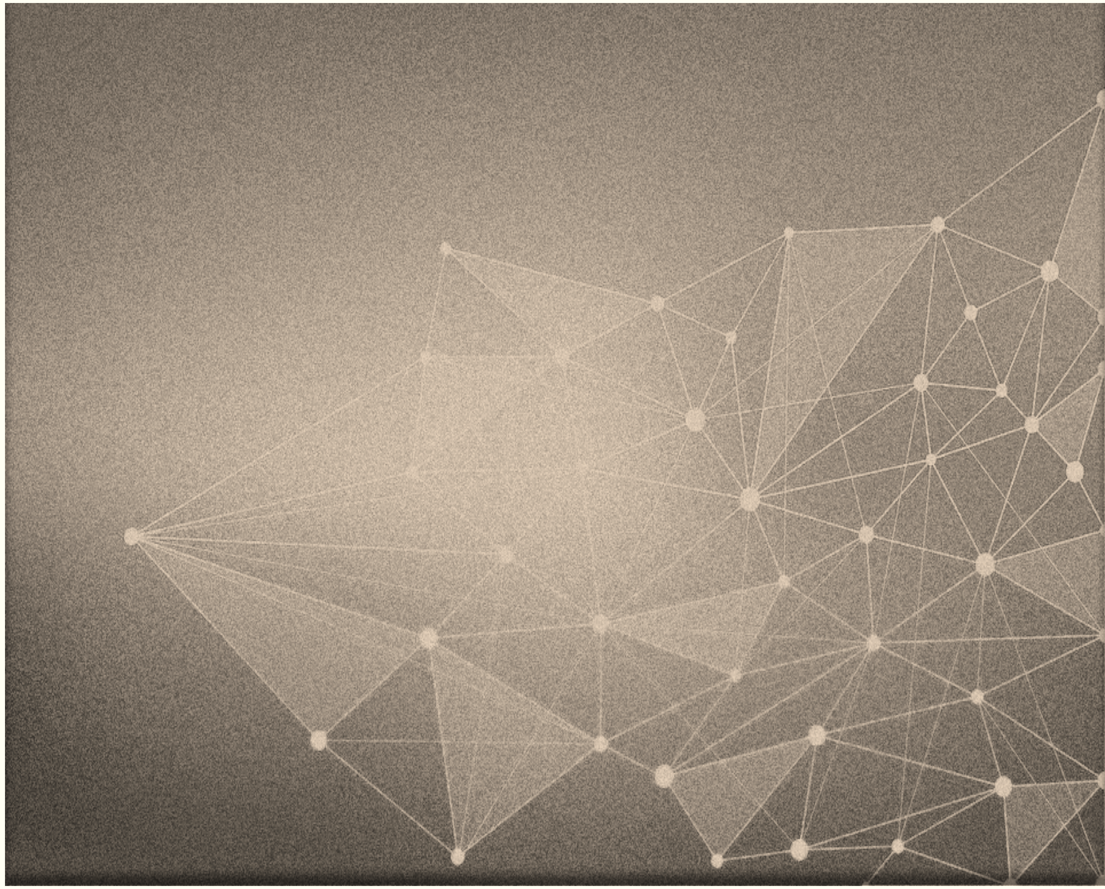
EECS2030:
ADVANCED OBJECT-ORIENTED PROGRAMMING

By: Dr. Marzieh Ahmadzadeh

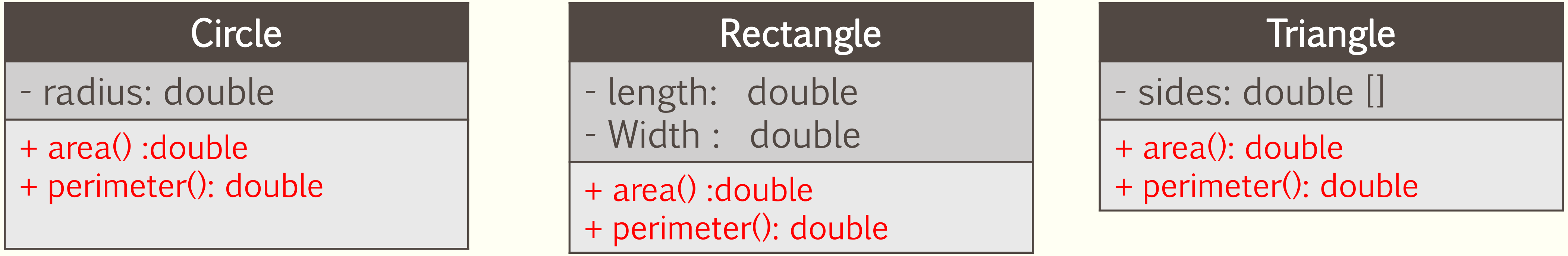


Outline

- Last Lecture:
  - Relationship between classes (Has-A)
    - Aggregation
    - Composition
- This week:
  - Another type of relationship: Is-A
    - Inheritance
    - Implementation
    - Overridden vs overloaded methods in inheritance relationship
  - Comparing Inheritance & Composition
  - Object Class

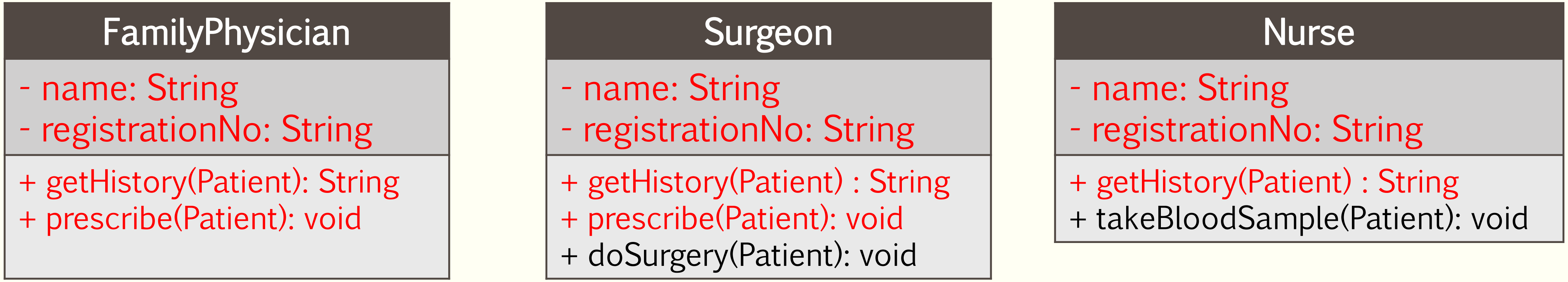
Inheritance

- It defines a relationship between objects.
- Different kinds of objects often have a **certain amount in common** with each other
- Example:



Inheritance

- It defines a relationship between objects.
- Different kinds of objects often have a **<u>certain amount in common</u>** with each other
- Example:



FamilyPhysician

- name: String
- registrationNo: String

+ getHistory(Patient): String
+ prescribe(Patient): void

Surgeon

- name: String
- registrationNo: String

+ getHistory(Patient) : String
+ prescribe(Patient): void
+ doSurgery(Patient): void

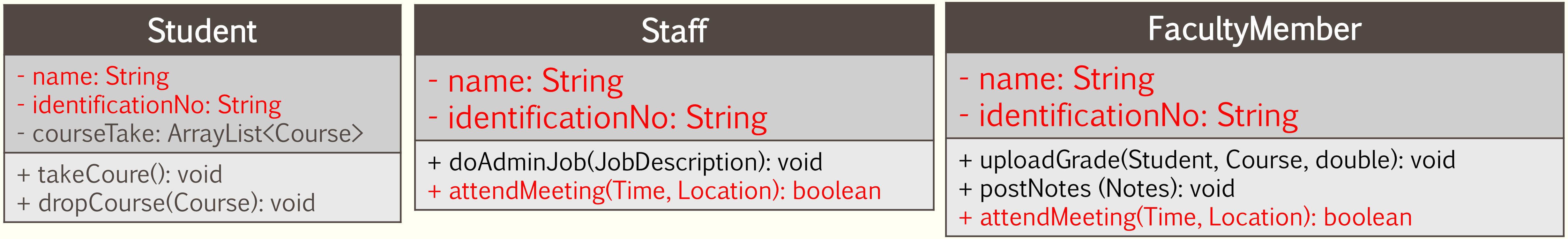
Nurse

- name: String
- registrationNo: String

+ getHistory(Patient) : String
+ takeBloodSample(Patient): void

Inheritance

- It defines a relationship between objects.
- Different kinds of objects often have a <u>certain amount in common</u> with each other
- Example:



Student

- name: String
- identificationNo: String
- courseTake: ArrayList<Course>

+ takeCoure(): void
+ dropCourse(Course): void

Staff

- name: String
- identificationNo: String

+ doAdminJob(JobDescription): void
+ attendMeeting(Time, Location): boolean

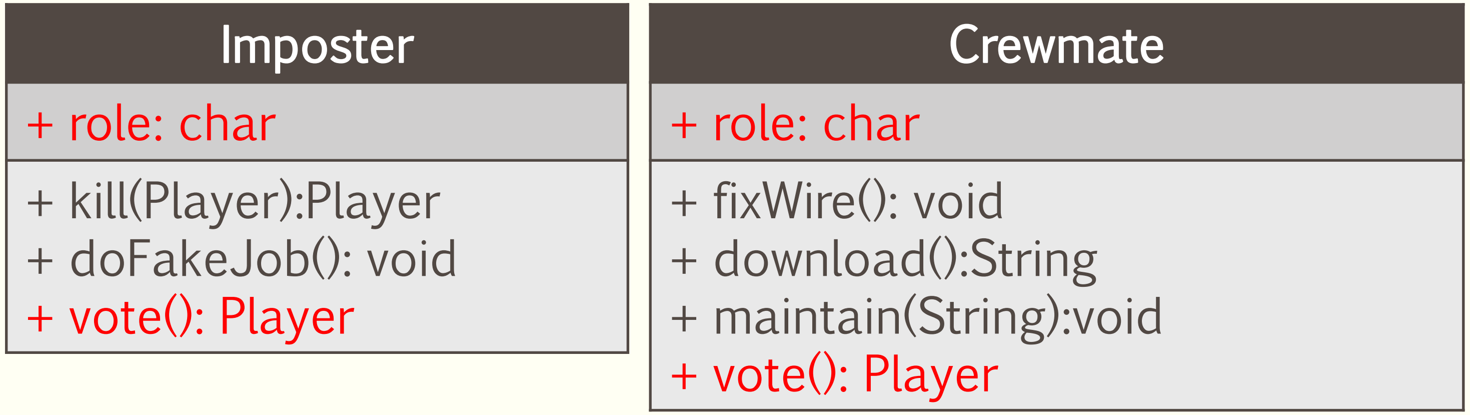
FacultyMember

- name: String
- identificationNo: String

+ uploadGrade(Student, Course, double): void
+ postNotes (Notes): void
+ attendMeeting(Time, Location): boolean

Inheritance

- It defines a relationship between objects.
- Different kinds of objects often have a **certain amount in common** with each other
- Example: Lab 4



Imposter

+ role: char

+ kill(Player):Player
+ doFakeJob(): void
+ vote(): Player

Crewmate

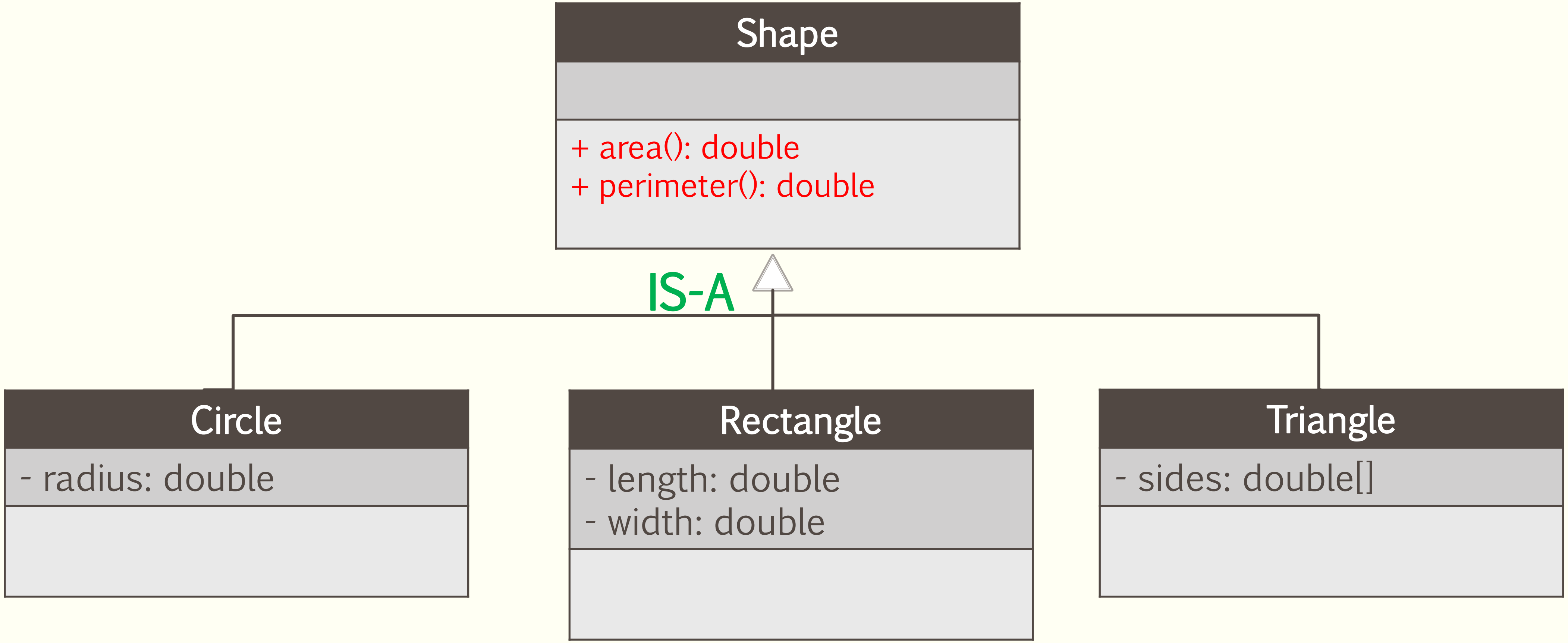
+ role: char

+ fixWire(): void
+ download():String
+ maintain(String):void
+ vote(): Player

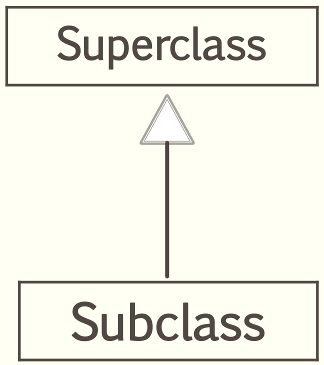
Inheritance; Code reuse

- By inheritance, you can write the `common code` once and use it as many times as required.
- The common codes are placed in a class that is called `superclass` or `base class`.
- The specific states and behaviors of an object stays in their own class, which is now is called `subclass` or `derived class`.
- When two (or more objects) share some attributes and methods, an IS-A relationship is created.
  - Subclass IS-A type of superclass
  - Circle and Shape, Student and University Member
- Subclasses inherits all the public and protected attributes and methods of their superclass.

Inheritance Example

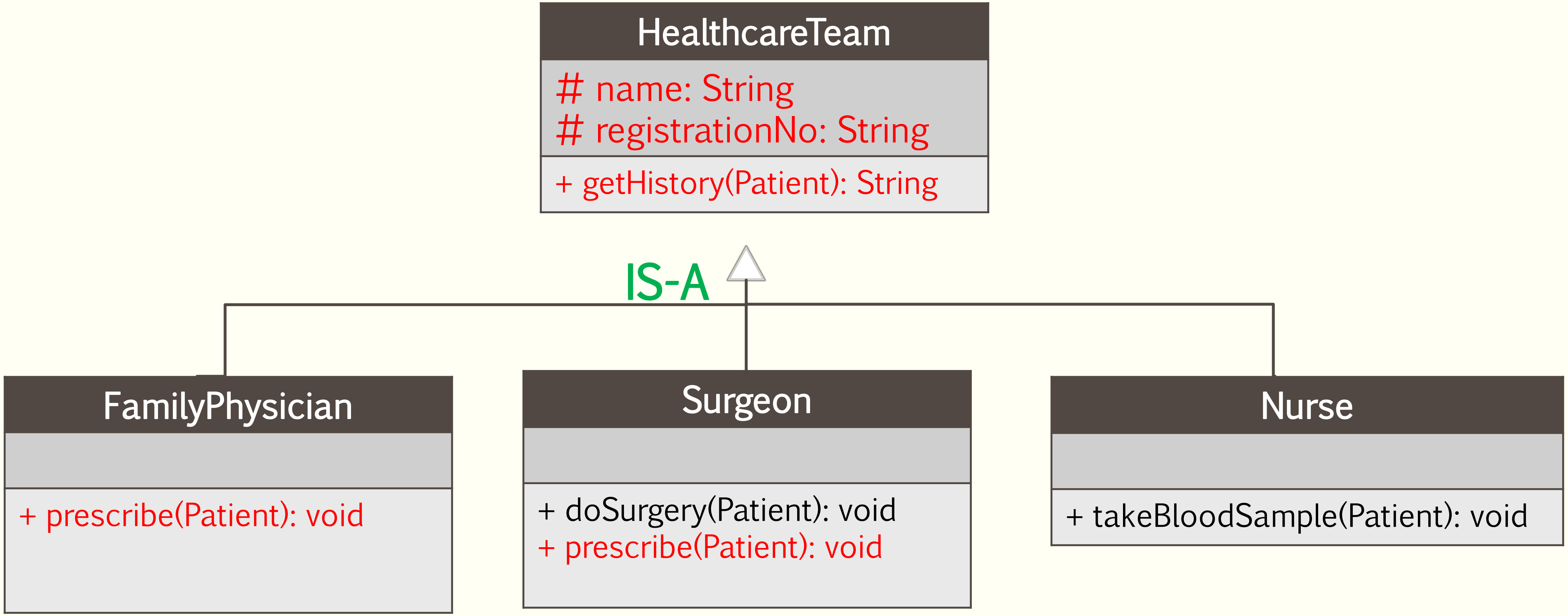


UML



- All the attributes and methods are written in UML as before.
- + and – is used for public and private access modifier.
- Protected features are shown with #.
- What is the difference between protected and private access modifiers?

More examples



HealthcareTeam

# name: String
# registrationNo: String

+ getHistory(Patient): String

IS-A

FamilyPhysician

+ prescribe(Patient): void

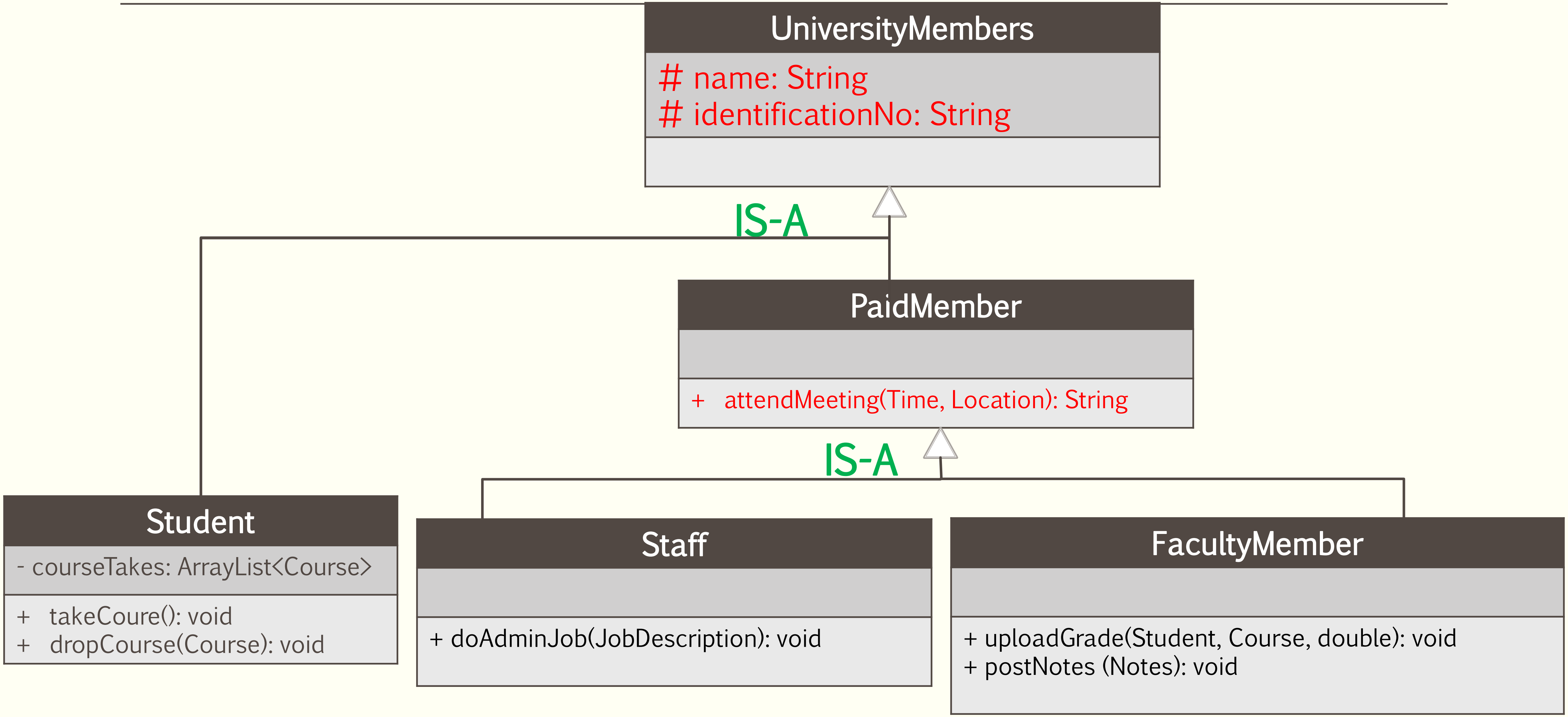
Surgeon

+ doSurgery(Patient): void
+ prescribe(Patient): void

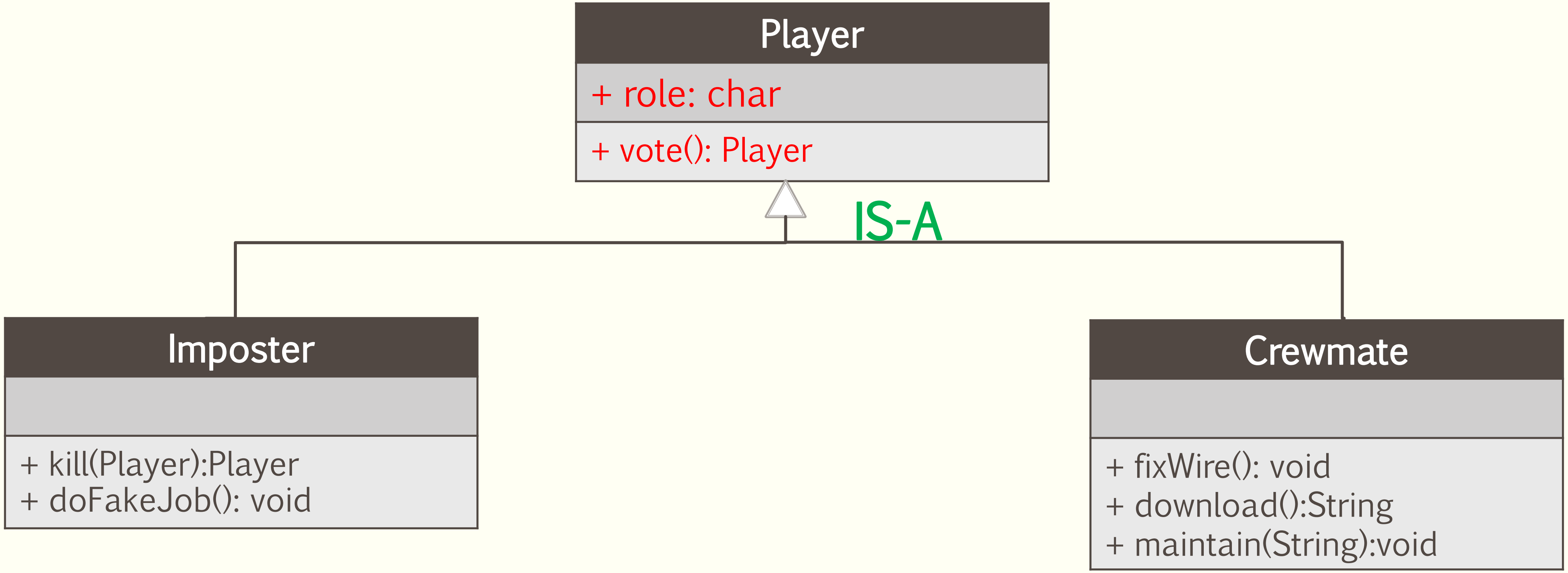
Nurse

+ takeBloodSample(Patient): void

Yet more example

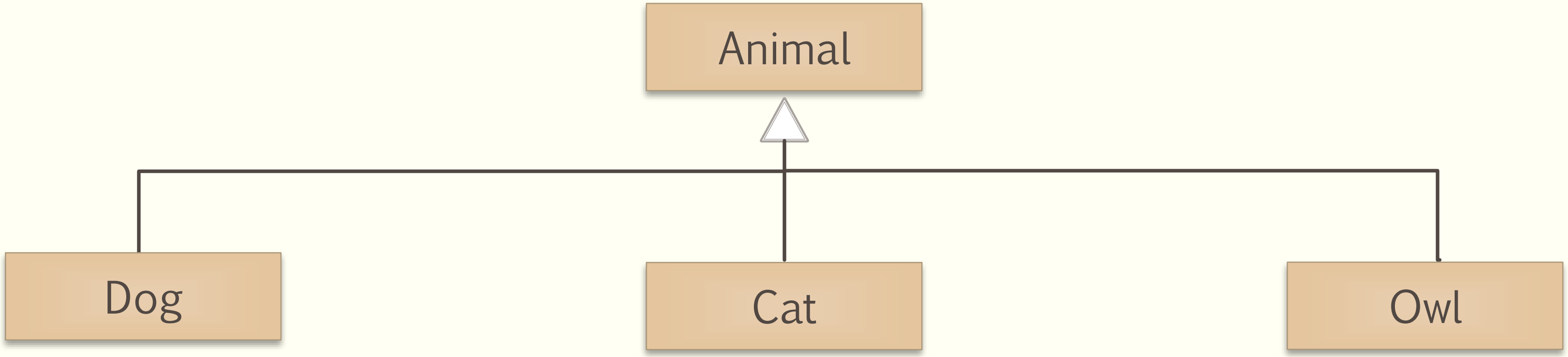

UniversityMembers
# name: String
# identificationNo: String
IS-A
PaidMember
+ attendMeeting(Time, Location): String
IS-A
Student
- courseTakes: ArrayList<Course>
+ takeCoure(): void
+ dropCourse(Course): void
Staff
+ doAdminJob(JobDescription): void
FacultyMember
+ uploadGrade(Student, Course, double): void
+ postNotes (Notes): void

Lab 4 Example:



Inheritance Syntax

- The subclass of a superclass `extends` it.

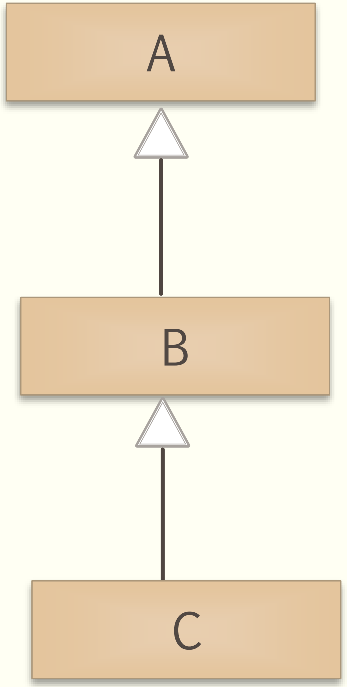


Inheritance; Constructors

- When there is a hierarchy of the classes and an object from the bottom of the hierarchy is constructed, the default constructors from the top to the bottom of the hierarchy get executed.
  - e.g. C object = new C()
  - e.g. C anotherObject = new C(object);

- In case it is required to call another constructor of the superclass, `super()` keyword is used.
  - Note: super() should be the first statement in the constructor.

- This is also called constructor chaining.



A

B

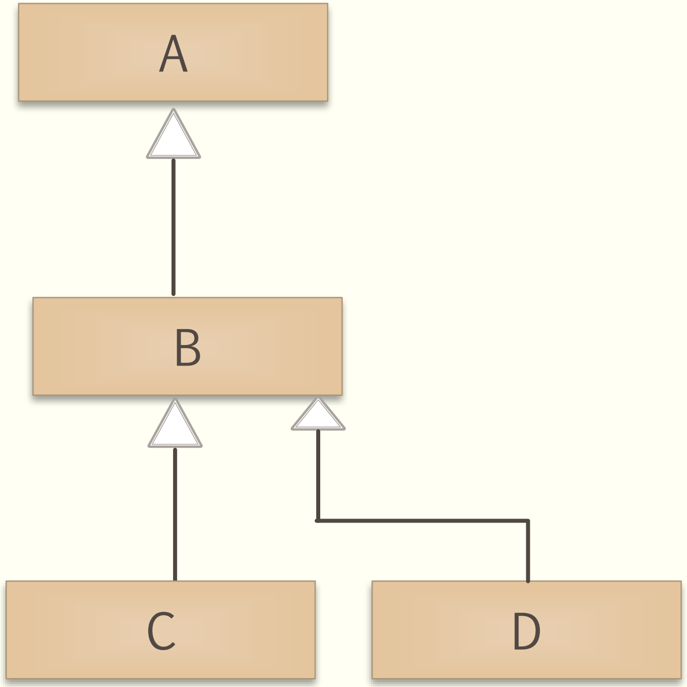
C

Questions

- Why constructor chaining happens when an object of the subclass is created?

- What is the difference between `this()` and `super()`.

- Can D call the constructor of C?



Activity

- A1 (Two Questions)
- A2 (One Question)

Inheritance; Overridden methods

- If a method in the parent class does not have the functionality that the child class is looking for, then the child class can override it.

- The overridden method has the same method signature as the original superclass method.
  - To avoid making a mistake use `@Override`

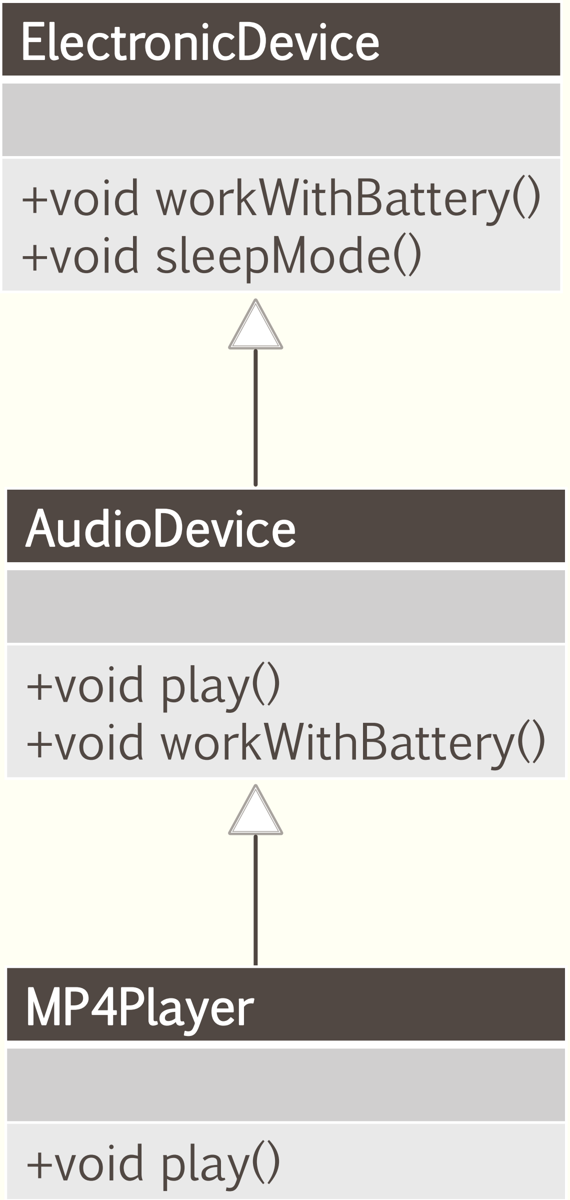
- In case more functionalities is to be added to the overridden method, you use super.*theMethodName* in the overridden method to take advantage of the functionality of the original method too.
  - The super keyword does not need to be the first line of the code.

Which method is called?

- Which method is called?

```
MP4Player myPlayer = new MP4Player();

myPlayer.play();

myPlayer.workWithBattery();

myPlayer.sleepMode();
```

- If a method is defined as a `final`, then it cannot be overridden.



How to recognise inheritance relation?

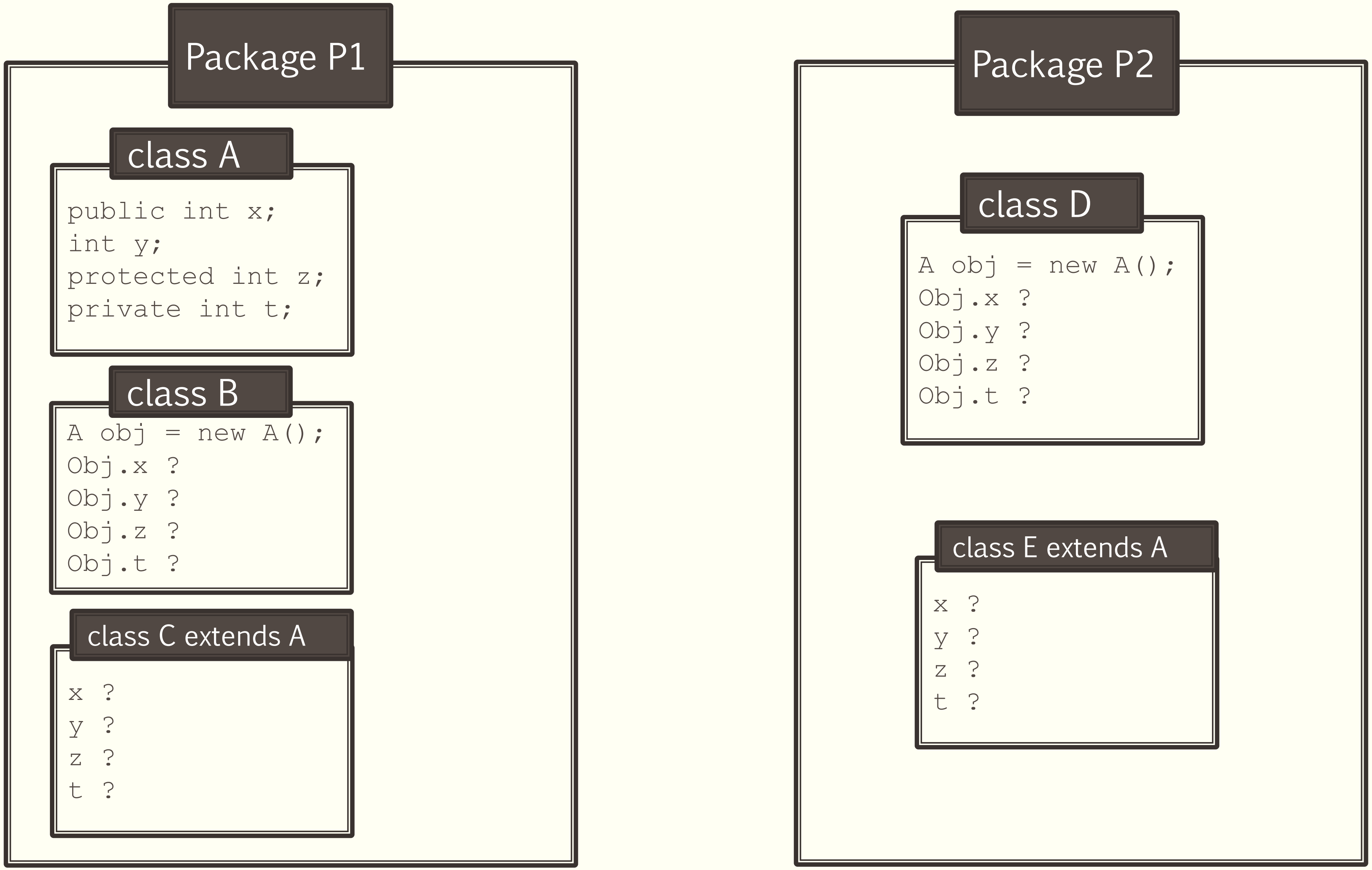
- There should be a IS-A relation for inheritance to work.
  - X IS-A Y means:
    - **X can do whatever Y can do** [even more] → Methods
    - **X can have whatever Y can have** [even more] → Instance variables
    - Therefore, X is a subclass and Y is a superclass.
  - Example:
    - A Surgeon IS-A Doctor therefore .....
    - A Square IS-A Shape therefore......
    - A Mountain bike IS-A Bicycle therefore......
    - A Dog IS-A(n) Animal therefore....
- Compare Is-A with Has-A.
  - Has-A: A class has a non-primitive instance variable
  - Is-A: subtype(subclass) is substitutable for the supertype(superclass)
    - Subtypes can satisfy supertypes' specification
    - Subtypes can do/have more than supertypes.

INHERITANCE & ACCESS MODIFIERS

What is inherited by the descendants?

- The descendants (subclasses) inherits all the ancestors' features whose access level is higher than (and including) protected access modifier.

Access Modifier	class	Subclass in the same package	package	Subclass in a different package	World (outside the package)
public	Y	Y	Y	Y	Y
protected	Y	Y	Y	Y	N
package (no A.M.)	Y	Y	Y	N	N
private	Y	N	N	N	N



Package P1

class A

```
public int x;
int y;
protected int z;
private int t;
```

class B

```
A obj = new A();
Obj.x ?
Obj.y ?
Obj.z ?
Obj.t ?
```

class C extends A

```
x ?
y ?
z ?
t ?
```

Package P2

class D

```
A obj = new A();
Obj.x ?
Obj.y ?
Obj.z ?
Obj.t ?
```

class E extends A

```
x ?
y ?
z ?
t ?
```

How to design (1)?

- First **find the common things**. (abstract characteristics that each object has)
  - e.g. all doctors have a name and they all treat patients.

- Design a class that represent those common states and behaviours. This forms the **superclass**.

- Decide if a **subclass** needs additional behaviours (method) or attributes that are specific to that particular subclass type.
  - e.g. a surgeon not only treat patients but also do surgery

- Draw the class hierarchy to make sense of what is inherited.
  - Superclass may have some features that should not be inherited. Make it private.
  - In the hierarchy, you may want to stop the inheritance, make the class 'final'.

How to design (2)?

- Sometimes inheritance is used when there is no is-A relationship.
  - The super and sublcass has many things in common.
    - Requirement: things that are not shared with the subclass must be private and do not get inherited.
  - OR The source code of superclass is hidden and
    - You need to override its method or add more methods and instance variables to better match your needs.
    - You should only use inheritance for this purpose, if ALL the instance variables and methods in the superclass makes sense to the objects that you want to create.
      - Stay tune for an example

Non-Extendable classes

- If a class is defined '**final**'. It means it is the end of inheritance.

- If a class is not defined as 'public', classes in different packages can not extend that class.

- If a class has only private constructors.

Overriding vs Overloading

- When you override a method, **the argument** and the **return type** should be the same as the method in supper class exactly .
    - Write `@Override` on top of the method.

- The access modifier in sub-method should be the same level or with higher access.

- If a method does not obey the above rules, it is not overridden anymore, it is **overloaded**.
    - In other words, overloaded methods are the methods that have the same name but perhaps different argument number, type or return type.
        - Overloaded methods that are different in 'return type' are definitely different in arguments list. But the reverse is not true.
        - Overloaded methods can have any access modifiers independent of the super method.

Summary

- A subclass extends a superclass
- A subclass inherits everything expect the private feature of the superclass.
- Inherited methods can be overridden.
  - Note: the lowest overridden method wins the call.
- Use the **IS-A** test to verify that the inheritance hierarchy is correct. If X extends Y, then X IS-A Y must make sense.
- The **IS-A relationship works one way** only. A dog is an animal but not all animals are a dog.

Activity

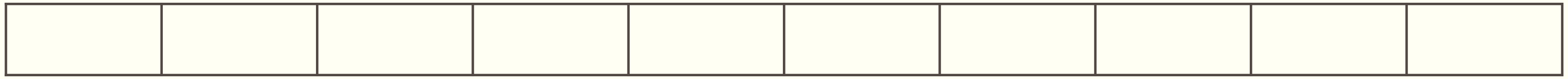
- Activity 3 (Q4) , Activity 4(Q5, Q5)

Recall

- Sometime inheritance is used not because there is an is-a relationship, but because you want to take advantage of the code that is ready to use.
    - This is not always a good idea.

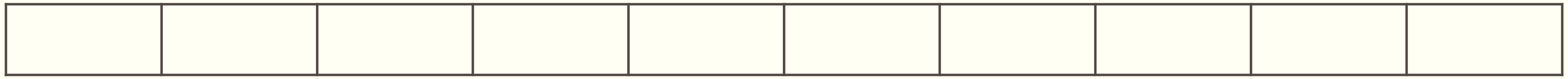
Queues

- A queue is a structure that is only accessible from its head [to remove an element] and from its tail to add new element.



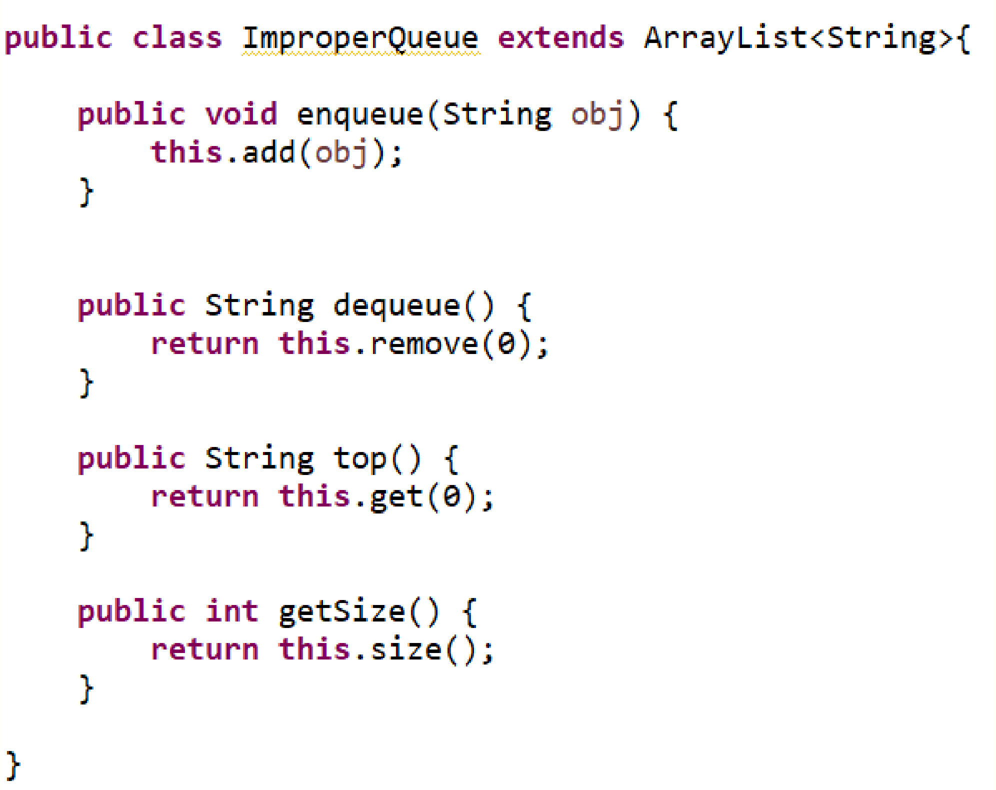
- A queue can be empty or have a large number of elements in it.

- Queues are very popular in computing. e.g. CPU Scheduling algorithms

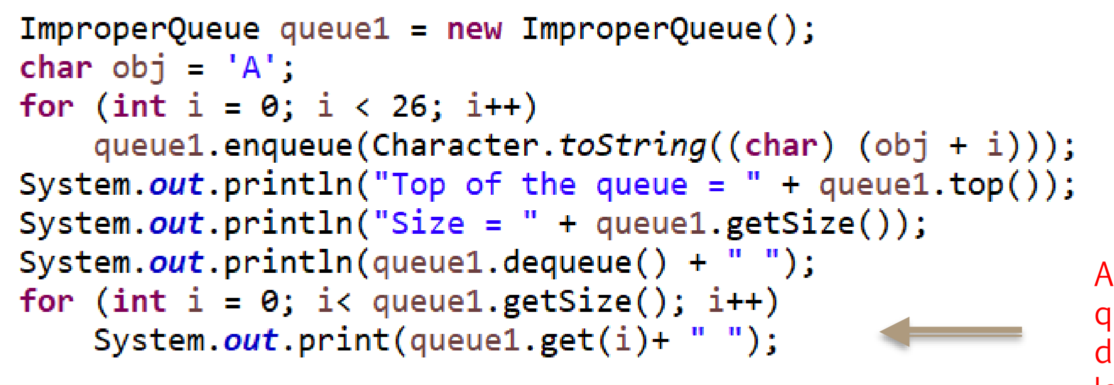


- Which of the following structures do you think is the most suitable to implement a queue if we don’t want to write the code from scratch?
  - Arrays
  - ArrayLists

Queues Implementation



```java
public class ImproperQueue extends ArrayList<String>{

    public void enqueue(String obj) {
        this.add(obj);
    }


    public String dequeue() {
        return this.remove(0);
    }

    public String top() {
        return this.get(0);
    }

    public int getSize() {
        return this.size();
    }

}
```

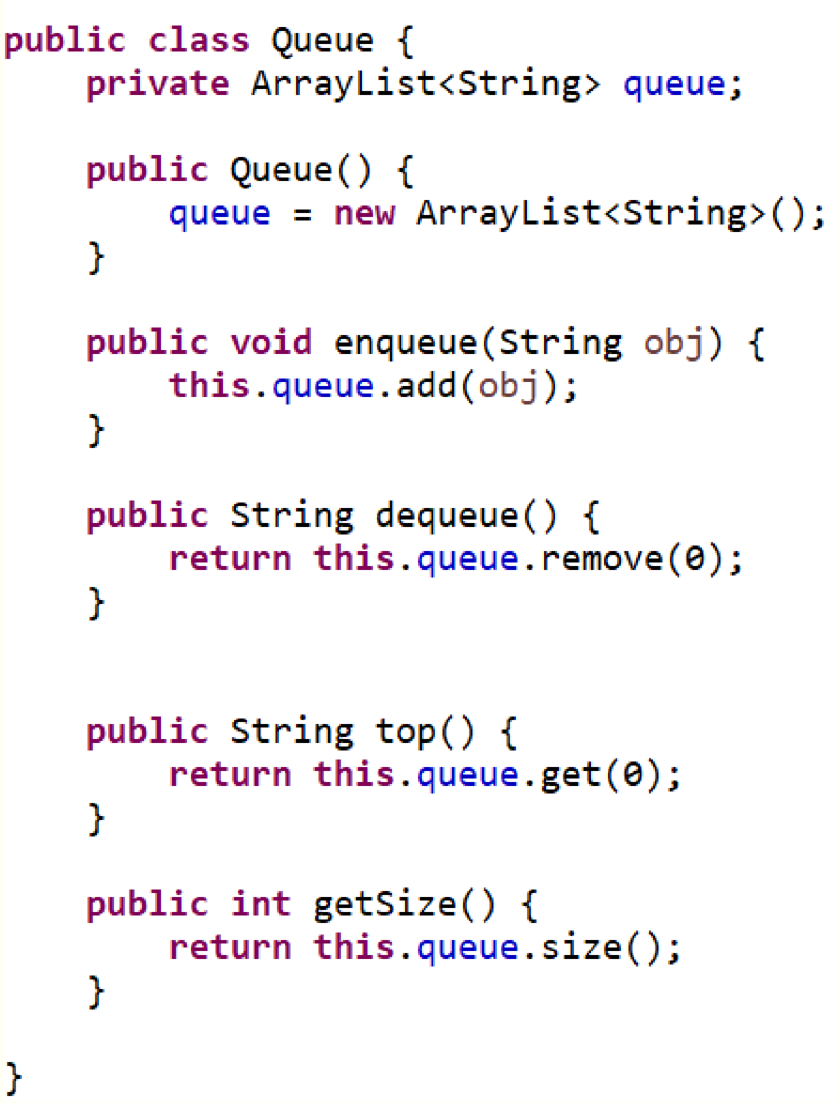


```java
ImproperQueue queue1 = new ImproperQueue();
char obj = 'A';
for (int i = 0; i < 26; i++)
    queue1.enqueue(Character.toString((char) (obj + i)));
System.out.println("Top of the queue = " + queue1.top());
System.out.println("Size = " + queue1.getSize());
System.out.println(queue1.dequeue() + " ");
for (int i = 0; i< queue1.getSize(); i++)
    System.out.print(queue1.get(i)+ " ");
```

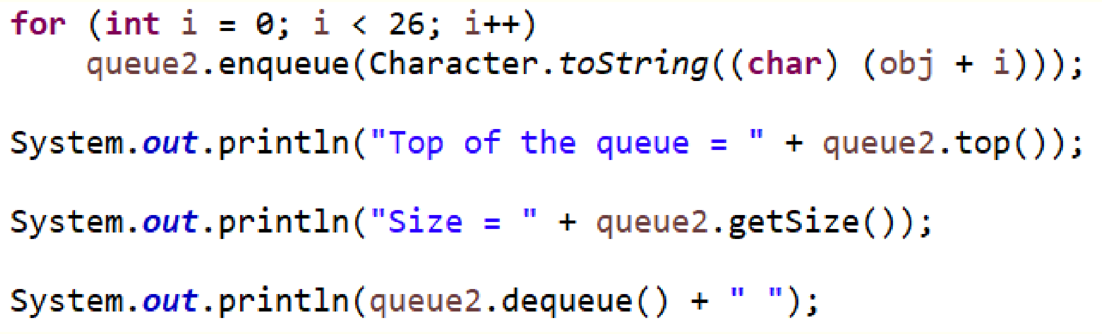
A proper queue does not let this happen

Implementing is-a is not always a good solution.
Instead use composition!

Queue: A Better Implementation



```
public class Queue {
    private ArrayList<String> queue;

    public Queue() {
        queue = new ArrayList<String>();
    }

    public void enqueue(String obj) {
        this.queue.add(obj);
    }

    public String dequeue() {
        return this.queue.remove(0);
    }


    public String top() {
        return this.queue.get(0);
    }

    public int getSize() {
        return this.queue.size();
    }

}
```



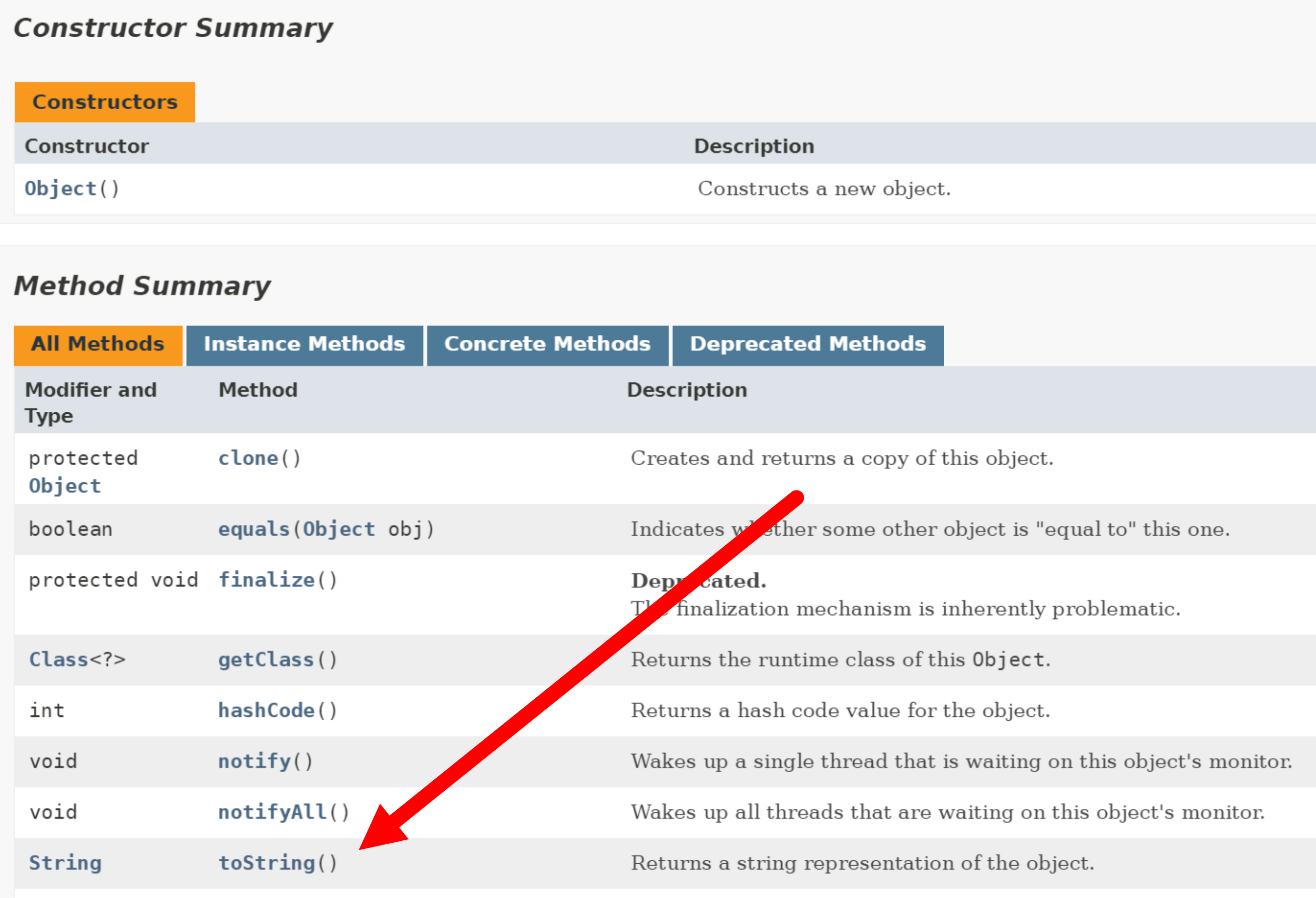
```
for (int i = 0; i < 26; i++)
    queue2.enqueue(Character.toString((char) (obj + i)));

System.out.println("Top of the queue = " + queue2.top());

System.out.println("Size = " + queue2.getSize());

System.out.println(queue2.dequeue() + " ");
```

There is no way that we can get access anywhere except the top of the queue for removal and back of the queue for insertion!

Class Object

- Object is a class that is the ancestor (superclass) of all the classes that are defined in Java.

- Now, the mystery of toString() method should be solved for you!



Constructor Summary

Constructors

Constructor	Description
`Object()`	Constructs a new object.

Method Summary

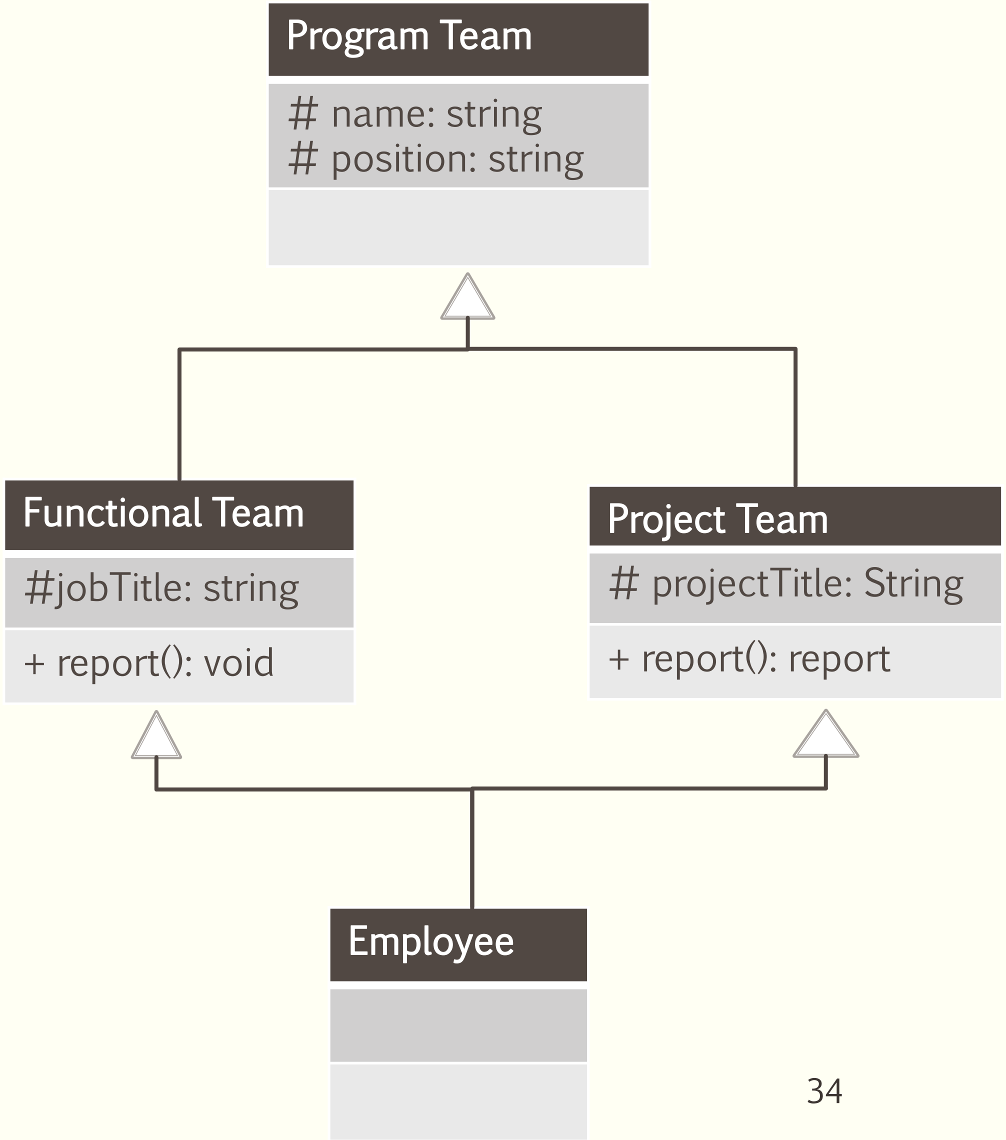
All Methods | **Instance Methods** | **Concrete Methods** | **Deprecated Methods**

Modifier and Type	Method	Description
`protected Object`	`clone()`	Creates and returns a copy of this object.
`boolean`	`equals(Object obj)`	Indicates whether some other object is "equal to" this one.
`protected void`	`finalize()`	**Deprecated.** The finalization mechanism is inherently problematic.
`Class<?>`	`getClass()`	Returns the runtime class of this `Object`.
`int`	`hashCode()`	Returns a hash code value for the object.
`void`	`notify()`	Wakes up a single thread that is waiting on this object's monitor.
`void`	`notifyAll()`	Wakes up all threads that are waiting on this object's monitor.
`String`	`toString()`	Returns a string representation of the object.

Single Inheritance

- Java does not allow `multiple inheritance` to avoid the problem of Deadly Diamond of Death.

```
Employee marzieh = new Employee();

marzieh.report();
```



Things to remember…

- In Java, each class is allowed to have **one direct superclass.**
- In Java each superclass has the potential to have an **unlimited number of *subclasses.***
- Superclass **does not know** of existence of any subclass.
  - Implications?
- If you change superclass, you need to compile the superclass only, without having to worry about the compiling of the subclasses again.
  - **Important:** You should not change any feature of the superclass that subclasses are dependant on. Like what?

Expectations & Reading

- Expectations
  - You should have a good understanding of what inheritance is, why it is required and how it is implemented.
  - You should be able to explain how an object is created when inheritance is involved.
  - You should be able to differentiate between overloaded and overridden methods.
  - You should be able to implement inheritance relationship for a given UML.
  - You should be able to think critically about the correctness of an inheritance.
  - You should be able to distinguish when to use composition and when to design a hierarchy of inheritance.
  - You should be able to solve the worksheet’s problem and explain the solution.
- Reading
  - Not required
- One Minute Paper