OBJECT ORIENTED PROGRAMMING LAB 7

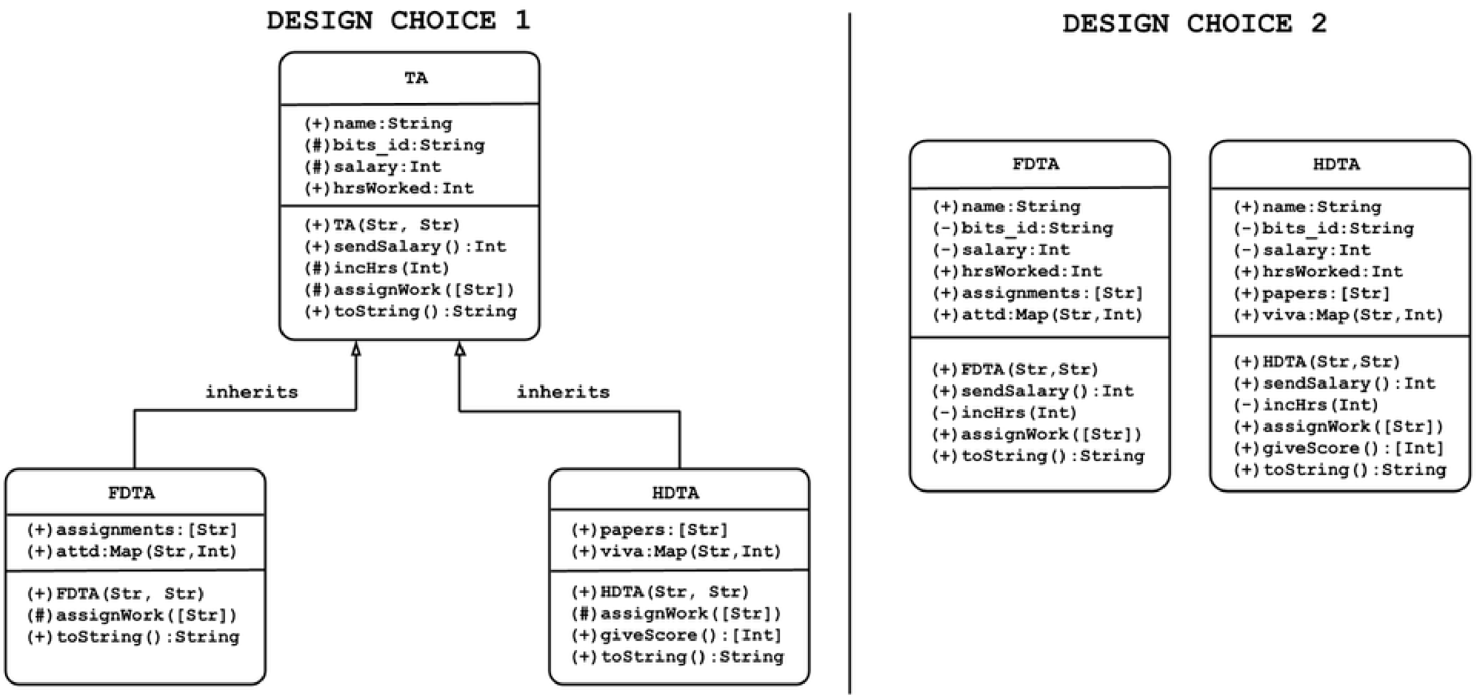
17th October 2023

Learning Objectives:

- Inheritance,
- Parent-Child Relationship Rules
- *IS A*, *HAS A* relationships
- Overriding
- Polymorphism

Inheritance

Inheritance is a mechanism by which a class can **inherit** (use) methods and variables of its parent class. The architecture of many languages is completely based on inheritance! For example in Java, every class by default extends the Object class. Why is inheritance important? Consider the following 2 designs choices:



Notice how the ***IS A*** relationship is implemented in Design Choice 1 (FDTA/HDTA ***is a*** Teaching Assistant). Which of the two designs is better? Choice 1! Because:

- **Reducing Redundancy/ Code Re-usability**- There are many common methods like *sendSalary()* and *incHrs()*. Inheritance allows us to write the code for these in the parent class Teaching Assistant and then use in FDTA and HDTA without the need to write the code again.

- **Ensuring Consistency** - What if the programmer forgets to implement the *sendSalary()* for FDTA in Design Choice 2? The FDTAs will work and get no money :(Inheritance helps in maintaining consistency across all classes of similar type. If any change needs to be made in the salary, it only needs to be done once in the TA class.

Other than these, it helps in making code modular, hierarchical, improves readability etc.

Implementation in JAVA

In Java, we use the ***extends*** keyword to implement inheritance (as you would have seen in the practice lab). An important feature of Java is that it allows inheriting only one parent, while languages like C++ allow a class to inherit from multiple parent classes.

In the previous labs, you have become familiar with ***this*** keyword. *this* allows us to refer to the methods/fields of object itself. Similarly, the ***super*** keyword allows us to refer to the methods and variables (provided they can be accessed) of the parent class.
Eg: super.sendSalary(), super.name. We can also call the constructor of the parent class within the constructor of the child class using *super* as you saw in the practice exercise. This is very useful and often used in inheritance.

The ***instanceof*** keyword allows us to check the type of a particular object. Very often the behaviour of the code changes when we are dealing with different children of the parent class. For example, if we want to issue an email to FDTAs, we can go through all TAs and check which of them are and *instanceof* FDTA and only include them.

To control the methods/ fields that the child class can use, we use the ***protected*** access modifier. We have already learnt about ***public*** and ***private*** access modifiers. *public* gives a complete access whereas *private* gives no access to outside classes. A *protected* method/field, on the other hand, can be accessed by the children of class but not other classes (outside the package). This gives us an intermediate level of protection required in inheritance. In UML diagrams protected is represented by a **#** (refer to the above diagram again!)

Overriding

The child classes can also override the behaviour of methods defined by the parent class. Refer to the TA example again. Notice the *assignWork()* method. even though it is implemented in Teaching Assistant, the FDTA and HDTA classes override it and re-define its behaviour (since FDTAs are given assignments to check while HDTAs correct midterms/compres). A child class can only override methods that are accessible to it! After overriding, the child class accesses its own code for the overridden method rather than its parent’s code. Overriding is quite useful. It allows us to re-define behaviour (if needed), otherwise use the original behaviour defined by the parent. This helps bringing in **customisation** without destroying consistency in the code.

Polymorphism

Polymorphism is one of the most important aspects of Object Oriented Paradigms. With polymorphism, an object of a given class can have multiple forms depending on its declared type or the kind of arguments used. It can be of 2 types - Static or Dynamic

Static (compile time) Polymorphism is resolved during compilation usually using overloading which you have learnt in previous labs.

Dynamic (Runtime) Polymorphism is resolved during runtime. It usually deals with method overriding and which method is executed depends on the type of the variable. We strongly encourage you to observe the behaviour of your code (by calling methods and variables of the object) with objects like:

```
Teaching Assitant t1 = new Teaching Assistant("John", "2021XXXXXX");
Teaching Assitant t2 = new FDTA("John", "2021XXXXXX");
Teaching Assitant t3 = new HDTA("John", "2021XXXXXX");
```

Lab Excercise Question

The Driver.java file has been implemented completely and you need to complete the rest of the 9 files. The specifications for each class are as follows:
(**Please complete the classes in the following order**)

- **Point2D** - Represents a coordinate in a 2D plane.
- **Shape** - Represents a general Shape.
- **Circle** - Represents a Circle in a 2D plane
  - **withinCircle** : Return true for all points that lie inside or on the Circle and false for the points that lie outside it.
  - **toString** : Return a String with all the necessary details about the Circle.

- **GameObject** - Represents a general object in the game. It can be a building or a character.
  - It has the following fields :
    1. **Circle range** : Represents the range of the object. The object is present at the centre of the Circle and can attack objects only within the Circle.
    2. **health** : Health may increase or decrease when other objects interact with the object. **If the health becomes 0, the object is dead**.
    3. **level** : Each object has various levels and the level may **increase** with time. **The levels start from 0.**
    4. **damage** : An int array representing the damage caused by the object at each level.

- Implementation details for some methods :
    1. **incLevel** : Increases the level of the object. Keep in mind that the level **should not become more than the max level.**
    2. **withinRange(GameObject obj)** : return true if obj is within the Range of this object, and false otherwise.
    3. **isDead**: return true if the object is dead, false otherwise.
    4. **getDamage**: return the damage that the object causes at its current level.
    5. **takeDamage**: Will be used when another object attacks this object.

- **Archer** - This is a character in the game. Complete the **attack** method. This is used by the Archer to attack another object. Keep in mind that an Archer **can attack any other object within its range** and the level of the Archer should increase after **every 2 successful attacks.** If we try to attack an object which is not in the range, print **"Out of Range".**

- **ArcherQueen** - This is a special Archer that does **twice the damage** as a normal Archer at each attack. Rest all features of the ArcherQueen are same as that of an Archer.

- **Cannon** - Represents a defence building in the game. Complete the **void attack(GameObject obj)** method. This is used by the Cannon to attack another object(obj). A Cannon must **not be able to attack other Cannons**. If we try to attack another Cannon, print **"Attack Failed"**. If we try to attack an object out of the Cannon's range, print **"Out of Range".** For any attack, only one of the two should be printed. First check for the Attack Failed condition. The level of the Cannon should increase after **every 4 successful attacks.**

- **Healer** - This is an object in the game that **heals other Archers.** The damage array of a Healer has **negative damages** and hence can be used to heal. If we try to heal a Cannon or another Healer, you should print **"Heal Failed"**. If we try to heal an Archer that is out of range, you should print **"Out of Range"**. The level of a Healer should increase after **every 3 heals**.

- **ArcherCannon** - This is **a Cannon which has an Archer** with it. Note that the ArcherCannon *HAS A* archer Both the Archer and Cannon **attack separately**. The Archer behaves like any other normal Archer. The **ArcherAttack** method is used to attack other objects using the Archer. The number of attacks and the level of the Cannon and the Archer are separate. The **isArcherDead()** method should return true if the Archer is dead. An Archer may die as a normal Archer dies or when the Cannon dies. i.e. the Cannon may live when the Archer is Dead but the Archer cannot live without the Cannon.

After completing each class, keep uncommenting the corresponding code in the Driver class to check if you have done it correctly. After finishing the lab, zip all the java files submit it on Quanta within time. No extra time will be given so submit your code even if incomplete.