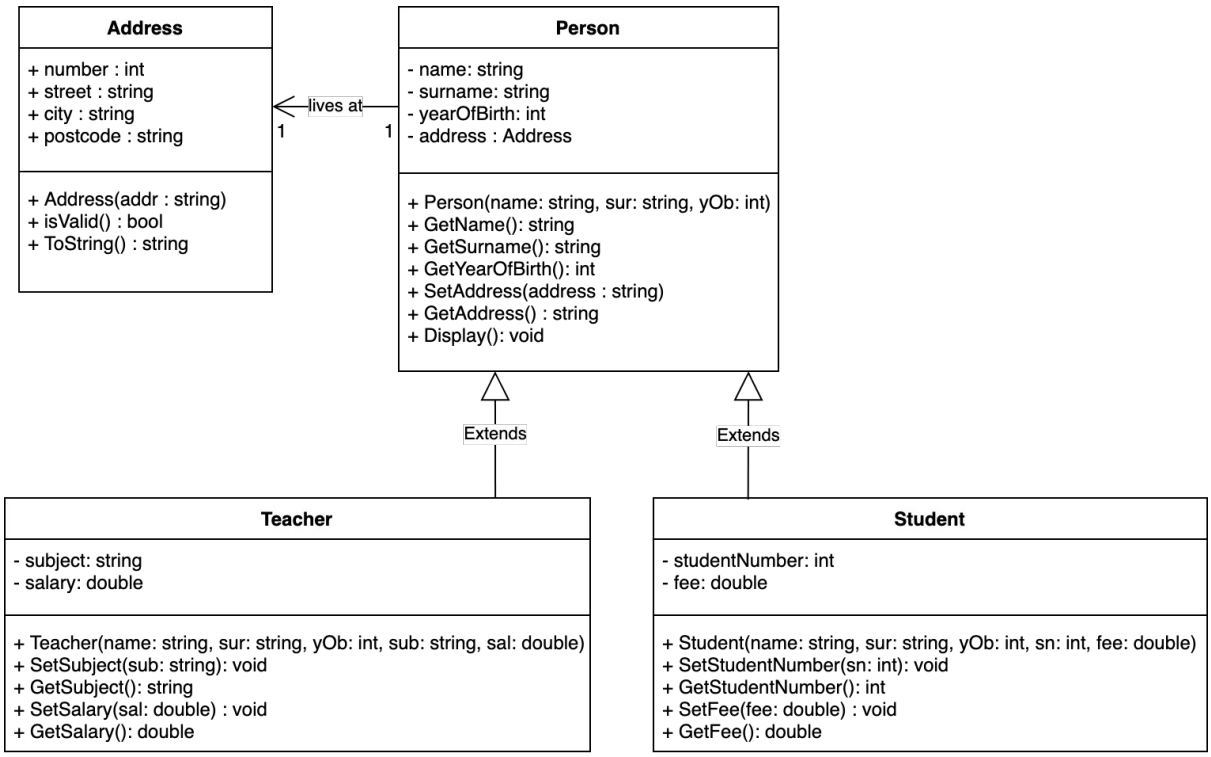
1: Generalisation relationship and Inheritance (with Tutor)

System Design

Develop a system to represent different types of people in a ***School*** according to the following UML Class Diagram:



The diagram shows a superclass (*parent* class) *Person* and two subclasses (*child* classes), *Student* and *Teacher*. A *Teacher is-a-kind-of Person* with additional properties, i.e., *salary* (double) and *subject* taught (e.g., "Computer Science", "Chemistry", etc.). A *Student* is a *Person* with the additional *fee* (the amount the student pays every year) and *studentNumber* attributes.

System Implementation

Start your implementation from the *Person* class developed during the tutorial of Week 8. The subclasses *Teacher* and *Student* should provide *getter* and *setter* methods for each of their specific attributes as represented in the diagram and detailed below.

Student Class

The *Student* class extends the parent class *Person* and has the following members:

- Two instance variables—*studentNumber* (int) and *fee* (double).
- One constructor to initialise *name*, *surname*, *yearOfBirth*, *studentNumber* and *fee*. The attributes inherited from the *Person* class should be initialised by referencing the superclass's constructor (use *base* for this).
- "setter" and "getter" methods—*SetStudentNumber*, *GetStudentNumber*, *SetFee* and *GetFee*.

Teacher Class

The *Teacher* class extends the parent class *Person* and has the following members:

- Two instance variables—*salary* (double) and *subject* (string).
- One constructor to initialise *name*, *surname*, *yearOfBirth*, *salary* and *subject*. The attributes inherited from the *Person* class should be initialised by referencing the superclass's constructor.
- "setter" and "getter" methods—*SetSalary*, *GetSalary*, *SetSubject* and *GetSubject*.

Program Class

Use the following *Program* class to test your system implementation.

```
using System;
using PersonProject;

namespace School
{
   class Program
   {
      static void Main(string[] args)
      {
         Person tom = new Person("Tom", "Jones", 1950);
         tom.SetAddress("30 Hampstead Ln; London; N6 4NX");
         Console.WriteLine("Surname: " + tom.GetSurname());
         Console.WriteLine();

         Student beth = new Student("Elisabeth", "Smith", 1995, 12345, 5000.0);
         beth.SetAddress("25 Castlegate; Knaresborough; HG5 8AR");
         Console.WriteLine("Surname: " + beth.GetSurname());
         Console.WriteLine("Fee paid: " + beth.GetFee());
         Console.WriteLine();

         Teacher sam = new Teacher("Sam", "Hamilton", 1970, 30000.0, "Computer Science");
         //sam.SetAddress("59 Pier Rd; Littlehampton; BN17 5LP");
         Console.WriteLine("Surname: " + sam.GetSurname());
         Console.WriteLine("Salary: " + sam.GetSalary());
      }
   }
}
```

Additional Questions

Question 1

Try to add the following statement at the end of the above *Main*:

```
Console.WriteLine("Salary: " + tom.GetSalary());
```

and explain what is happening.

Question 2

As you know, a *Display* method is defined in the *Person* class. What happens if you invoke that method on a *Student* or *Teacher* object? Think about a possible way to solve what you see.

2: Object Relationships (independent work—pair programming)

I invite you today to work in pairs using **Pair Programming**—an agile software development process in which two programmers work together at one workstation. One, the *driver*, writes code while the other, the *observer* or navigator, reviews each line of code as it is typed in. **The two programmers switch roles frequently.**

While reviewing, the observer also considers the "strategic" direction of the work, coming up with ideas for improvements and likely future problems to address. This is intended to free the driver to focus on the "tactical" aspects of completing the current task, using the observer as a safety net and guide.

System Design

Draw a UML class diagram to model information about *Movies* and *Actors* (you can use https://app.diagrams.net/?src=about for this).

Movie Class

Start your design by adding to the diagram a *Movie* class that includes the fields described below.

- Three *private instance variables,* i.e., the *title* of the movie (string), the *category* (string) and the *number of awards* (int).
- One *constructor* to initialise the *title* and *category* attributes
  - public Movie (string title, string category) { … }
- Public *getters* and *setters*:
  - public string GetTitle(){ … }
  - public string GetCategory() { … }
  - public void SetNumAwards(int numAwards) { … }
  - public int GetNumAwards() { … }

Actor Class

Now add an *Actor* class to the diagram. Try to identify relevant instance variables, constructors and setter and getter methods for this class. An *Actor* object has the *name*, *surname*, and *yearOfBirth* attributes, as well as information about the total *number of movies* and the *annual salary*. These last two attributes are not initialised via the constructor but the corresponding setter methods.

Relationship between objects

As discussed during the Lecture (Week 9), one object can be related to other objects and use their functionalities. Identify the type of relationship (association, aggregation, composition, etc.) between instances of the *Movie* and *Actor* classes and the multiplicity; then add the relationship to your diagram.

System Implementation

Translate the UML Class Diagram that describes the system's design into the corresponding C# code. When implementing the relationship between the two classes, please consider that a *Movie* object can be related with a maximum of 100 *Actor* objects. Test your implementation by using the following *Program* class.

Program Class

```
using System;

namespace Movies
{
   class Program
   {
      static void Main(string[] args)
      {
         Actor johnny = new Actor("Johnny", "Depp", 1963);
         Actor keira = new Actor("Keira", "Knightley", 1985);

         johnny.SetNumOfMovies(100);
         johnny.SetSalary(400000);

         keira.SetNumOfMovies(50);
         keira.SetSalary(300000);

         Movie movie = new Movie("Pirates of the Caribbean", "Adventure");

         if (!movie.AddActor(johnny))
            Console.WriteLine("Error adding actor");

         if (!movie.AddActor(keira))
            Console.WriteLine("Error adding actor");

         movie.SetNumAwards(10);

         movie.Display();
      }
   }
}
```

Output
Title: Pirates of the Caribbean
Category: Adventure
Number of Awards: 10
Number of Actors: 2

Actor 1
Name: Johnny
Surname: Depp
Year of Birth: 1963
Number of movies: 100
Salary: 400000

Actor 2
Name: Keira
Surname: Knightley
Year of Birth: 1985
Number of movies: 50
Salary: 300000

3: Inheritance and code reuse (at home)

You may have noticed that the *Actor* class has features (attributes and behaviours) similar to the *Person* class implemented in exercise 1. You can take advantage of *inheritance* and reuse existing code from the *Person* class!