# Lecture 16 - The OO Paradigm - Abstract Data Types, Information Hiding and Objects

Collin Roberts

November 6, 2023

# Outline

# Abstract Data Types (§7.5)

## Definition 1

*An* **abstract data type** *is a mathematical model of*

1. *the data objects comprising a data type, and*
2. *the functions that operate on these data objects.*

# Examples

1. The **integers** are an ADT, defined as the values $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$, and by the operations of $+$, $-$, $*$, and sometimes $/$, etc., which behave according to the familiar rules of arithmetic (e.g. associativity, commutativity, distributive laws, no dividing by 0, etc). Typically integers are represented in a data structure as binary numbers, but there are many representations. The user is abstracted from the concrete choice of representation, and can simply use the data objects and operations according to the familiar rules.
2. a **stack** (i.e. a last-in, first-out data structure).

# Remarks

1. An **abstract data type (ADT)** need not be an arithmetic object itself; however each of its operation must be defined by some **algorithm**.
2. In CS, an **abstract data type (ADT)** is a mathematical model, where a **data type** is defined by its behaviour ("what it does", not "how it does it") from the point of view of a user (not an implementer), specifically:
    1. possible values,
    2. possible operations on data of this type, and
    3. the behaviour of these operations.

# Remarks

3. This contrasts with **data structures**, which are concrete representations of data, from the point of view of an implementer, not a user.
4. Using abstract data types supports abstraction of both kinds, data and procedural.
5. Hence abstract data types are desirable from the viewpoints of both development and maintenance.

# Information Hiding (§7.6)

This is the last key ingredient in understanding the OO paradigm.

## Definition 2

**Information hiding** *means hiding the implementation details of a module (data + code) from the outside world.*

# How Information Hiding is Useful at Design Time

1. Make a list of implementation decisions which are likely to change in the future.
2. Design the resulting modules such that these implementation details are **hidden** from other modules.
3. This practice protects other parts of the software product from the impact of extensive changes if the implementation decisions are changed.

# Remarks

1. A module affords this protection by
   1. encapsulating the data/operations to be hidden together,
   2. hiding the details using a language construct like `private`, and
   3. providing a stable **interface**.
2. A **class** (Definition 3) may be implemented
   1. without information hiding (bad), or
   2. with information hiding (good).

# Remarks

1. The text attempts to exhibit a "straight line" path from modules to objects. In my humble opinion this does not tell the OO story correctly.
2. All ingredients need to be (independently) done well for effective OO development, which will realize the benefits of:
    1. fewer regression faults,
    2. cheaper maintenance and
    3. re-use.
3. **Reminder:** Use our definitions from the Lectures Notes instead of the definitions from the text, where there are any conflicts.

# Classes

## Definition 3

*A* **class** *is an* **abstract data type** *(Definition 1) that supports* **inheritance** *(Definition 4).*

## Definition 4

**Inheritance** *allows a new data type to be defined as an extension of a previously defined type, rather than having to be defined from scratch.*

# Examples

(Remark: the text example, in which `Person` is the parent class of the `Parent` class, is needlessly confusing!)

1. Start with a `Person` class, having
    1. **Properties (/ Attributes)**
        1. `LastName`,
        2. `FirstName`,
        3. `DateOfBirth`

    and
    2. **Methods**
        1. `createFullName`,
        2. `createEmail` and
        3. `computeAge`.

# Examples

2. Then define a `Student` class, having all the **Properties/Methods** of `Person`, plus
    1. **Properties**
        1. `StudentNumber`
        2. `CumulativeAverage` (in reality we would compute this from individual grades rather than storing it; we make it a property here for simplicity).

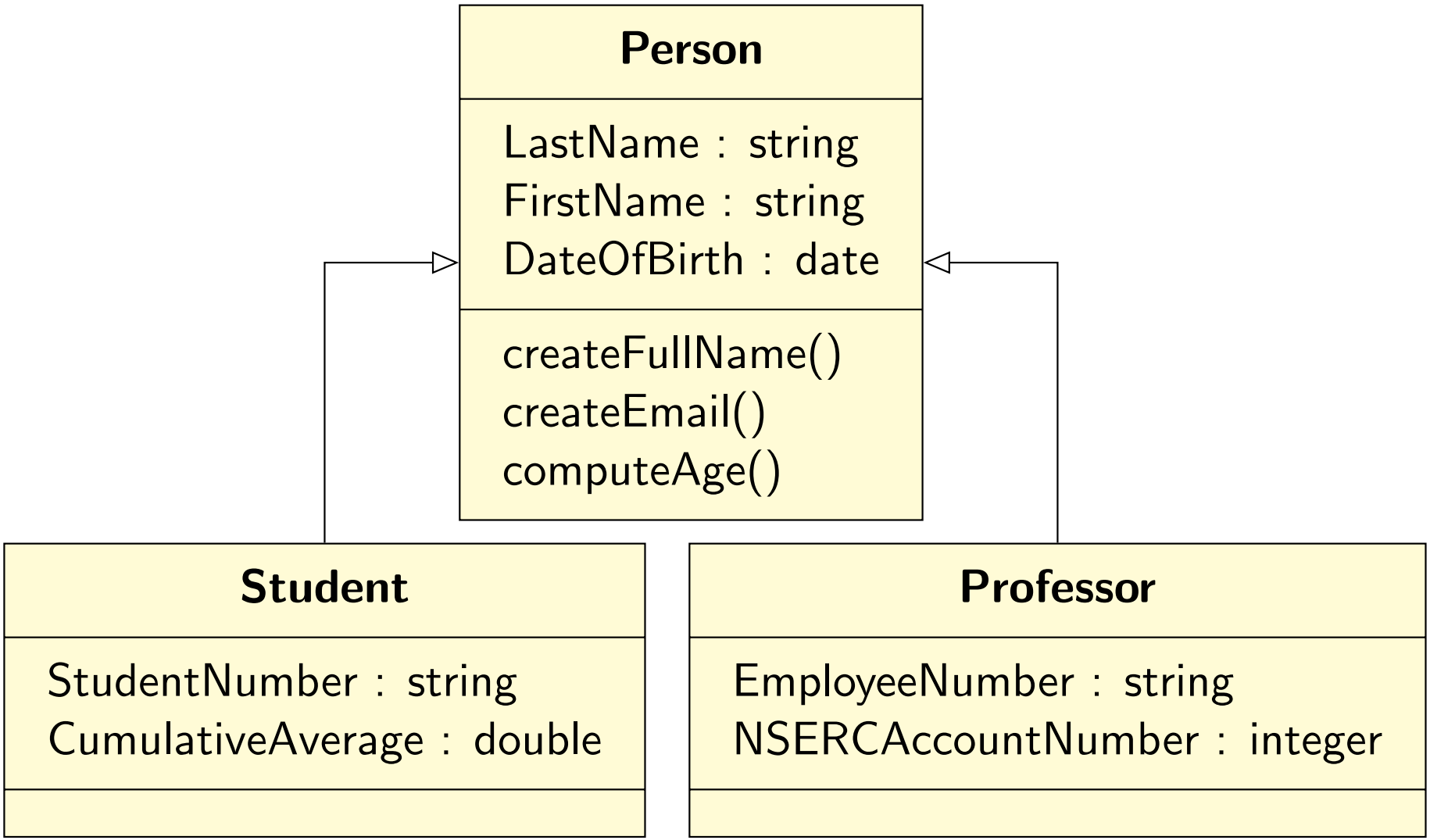
# Examples

3. Then define a `Professor` class, having all the **Properties/Methods** of `Person`, plus
    1. **Properties**
        1. `EmployeeNumber`
        2. `NSERCAccountNumber`.

# Examples

4. Then each of `Student`, `Professor`
    1. **inherits** from `Person`,
    2. **isA** `Person`, and
    3. is a **specialization** of `Person`.

Here is a diagram of the relationships between these classes.



# Objects

## Definition 5

*An* **object** *is an instantiation of a class (Definition 3).*

# Examples

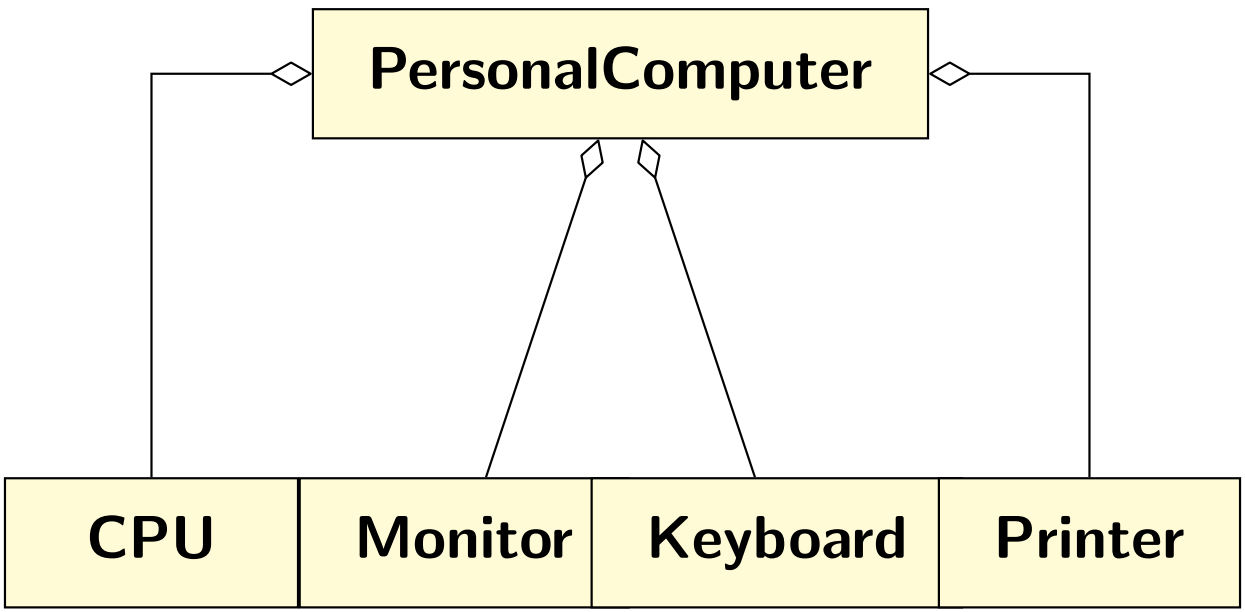
1. `CollinRoberts` could be an instantiation of the `Professor` class.

# Aggregation

## Definition 6

**Aggregation/Composition** *refers to the component classes of a larger class (i.e. grouping related classes creates a larger class).*

# Aggregation Examples

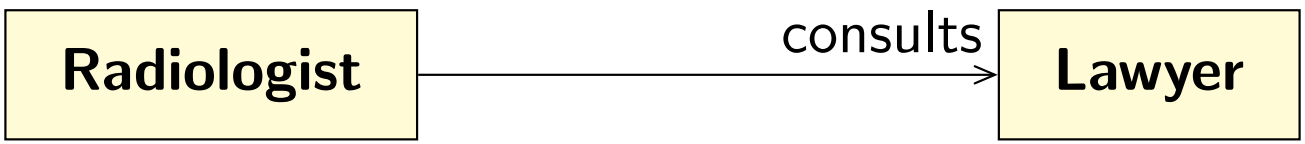


# Association

## Definition 7

**Association** *refers to a relationship (of some kind) between two apparently unrelated classes.*

# Association Examples




The diagram indicates that `Radiologist` consults `Lawyer`.