Exercise 8.1 – Inheritance

- The goal of this exercise is to write a class representing a Manager through inheritance from an existing class Employee
- Approach – writing class Manager
  a) Copy the file `ex8.1/Employee.hh`, look at the class and understand all the features.
  b) Create a small main program, instantiate an `Employee` and print it information by calling its `businessCard()` function. Also print out the salary information accessed through the `salary()` member function in the main program.
  c) Write a new `class Manager` that inherits from `class Employee`. The class Manager should have the following additional data members.
     - `set<Employee*> subordinates` – A set of subordinates that the manager manages

     Why is a *set* a better choice than e.g. a *list* or *vector* to keep track of a collection of subordinates?
  d) Add the following accessor/modifier functions for the employee set of class Manager
     - `void addSubordinate(Employee& empl)` – A method that adds a subordinate employee to the subordinates list ;
     - `const set<Employee*>& listOfSubordinates() const` – A method that returns a const reference to the employee managed by this manager

Exercise 8.1 – Inheritance

e) Extend your main program such that it also creates a `Manager` object. Print the manager information using the `businessCard()` function and also print its salary information retrieved through `salary()`.

f) Does in your opinion the manager object behave exactly like the employee object as long as you only refer to the employee-defined properties of both (such as the business card)?

- Approach – Write a better business card method for `class Manager`

g) Implement the function `void businessCard(ostream& os = cout) const` in class Manager. The idea is that this business card function writes a better version that supersedes the employee-style business card.

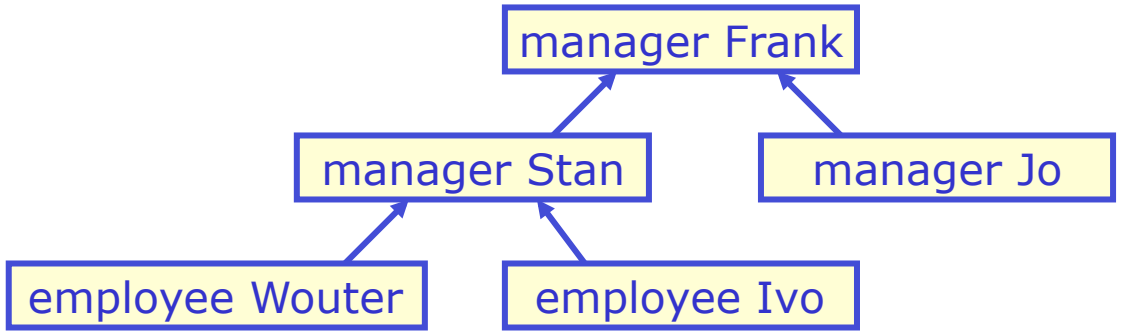
h) The idea is that the manager-style card will be like the employee-style card *plus* some extra information. So we first call the employee business card function inside the manager business card function. The name of the employee-style function is the same as that of the manager-style function, i.e. `businessCard()`, but use the scope operator we can be specific: call `Employee::businessCard()` within the `businessCard()` implementation of `Manager`. After that call, add code to the `businessCard()` implementation of `Manager` that prints the set of managed employees. Use an `iterator` to go through the set of employees.

i) Rerun the `main` program and see if the managers business card is printed out in the new style

Exercise 8.1 – Inheritance

- Approach – Creating a hierarchy of manager and employees

  j) Enter the following personnel hierarchy in the main program



  k) Use classes `Employee`, `Manager`, and function `Manager::addSubordinate()`

  l) Do you have problems entering a `Manager` as `Employee` in function `Manager::addSubordinate()`? Why is(n't) that?

  m) Print out everybody's business card to verify that you entered the personnel hierarchy correctly

© 2006 Wouter Verkerke, NIKHEF

Exercise 8.2 – Polymorphism

- The goal of this exercise is to introduce polymorphism, i.e. make managers behave like managers, even when addressing through an `Employee*` pointer
- Approach – Organizing your directory of business cards
  a) Start with the output of Ex8.1. First we will split our main program in two parts: one part that creates all the employees and managers, and a second part that receives a list of all employees and managers stored in a `set<Employee*>`. The goal of this reorganization is to be in a position where a part of our code doesn't really know if a given employee (a member of) is really an employee or a manager seen as employee.
  b) Modify the main program such that it stores pointers to all employees and manager into `set<Employee*>` after all of them have been created.
  c) Remove from `main()` the code that prints everybody's business card. Instead write a global function `printAllCards(set<Employee*> directory)` that prints everybody's business card and call the function from `main()`.
  d) Look carefully at the printout of all business cards. Does the output differ from that of Ex 8.1? Explain the difference.
  e) Modify `printAllCards` such that it takes two const iterators: `begin` and `end` instead of the set.

© 2006 Wouter Verkerke, NIKHEF

Exercise 8.2 – Polymorphism

- Approach – Changing `Employee` into a polymorphic type

  f) What have we learned from this exercise

  - We have seen that inheritance allows you to build a `Manager` out of an `Employee` and that a `Manager` can redefine the behavior that was originally implemented by `Employee`, such as the implementation of the `businessCard()` function. The drawback of this approach is that it only works if we (the caller of `businessCard()`) know if a given object is an `Employee` or `Manager`. If we don't and, as done before, address all individuals like `Employees`, they will all behave like `Employees`.
  - What is more desirable in many circumstances (and in ours) is that `businessCard()` function is an *abstract interface*, i.e. it defines *how* we can have a certain action performed from an employee, but that the exact action that follows depends on what each object knows about itself. That means that a Manager would always print a manager-style business card, even through we addressed it through an `Employee*` pointer.

  g) Add keyword `virtual` in front of `Employee::businessCard()`. This change introduces polymorphic behavior. Print out the directory of business cards again. Does it look different now?

© 2006 Wouter Verkerke, NIKHEF

Exercise 8.3 – Abstract base classes

- The goal of this exercise is to manipulate a collection of different types of shapes through a common base class Shape
- Approach – Creating a class Circle
  a) Look at the abstract base class `Shape` in `ex8.3/Shape.hh` and at the implementation of class Square in `ex8.3/Square.hh`. Write a small main program that creates a `Square` object and prints out its surface and circumference.
  b) Write a class `Circle`, similar to class `Square`, that inherits from class `Shape`. Class `Circle` should have one data member radius that should be initialized in the constructor `Circle::Circle(int radius)`
  c) Adapt the main program to also create a `Circle` object and print out its surface and circumference too.
- Approach – Using and creating polymorphic lists
  d) In the main program create a `list<Shape*>`. Next, create several `Circle` and `Square` objects and add pointers to those objects to that list.
  e) Write a `void listShapes(list<Shape*> l)` function that prints the surface and circumference of all objects in the shape list using the virtual member functions `surface()` and `circumference()`.

© 2006 Wouter Verkerke, NIKHEF

Exercise 8.3 – Abstract base classes

- Approach – Extending the `Shape` interface

  f) Add a pure virtual function `const char* shapeName() const = 0` to class `Shape` and recompile your code. Does it compile OK?

  g) Add the *implementations* of `shapeName()` to `Square` and `Circle` and compile and run again.

  h) Modify the `listShapes()` routine to also print out the shape name of each shape.

© 2006 Wouter Verkerke, NIKHEF