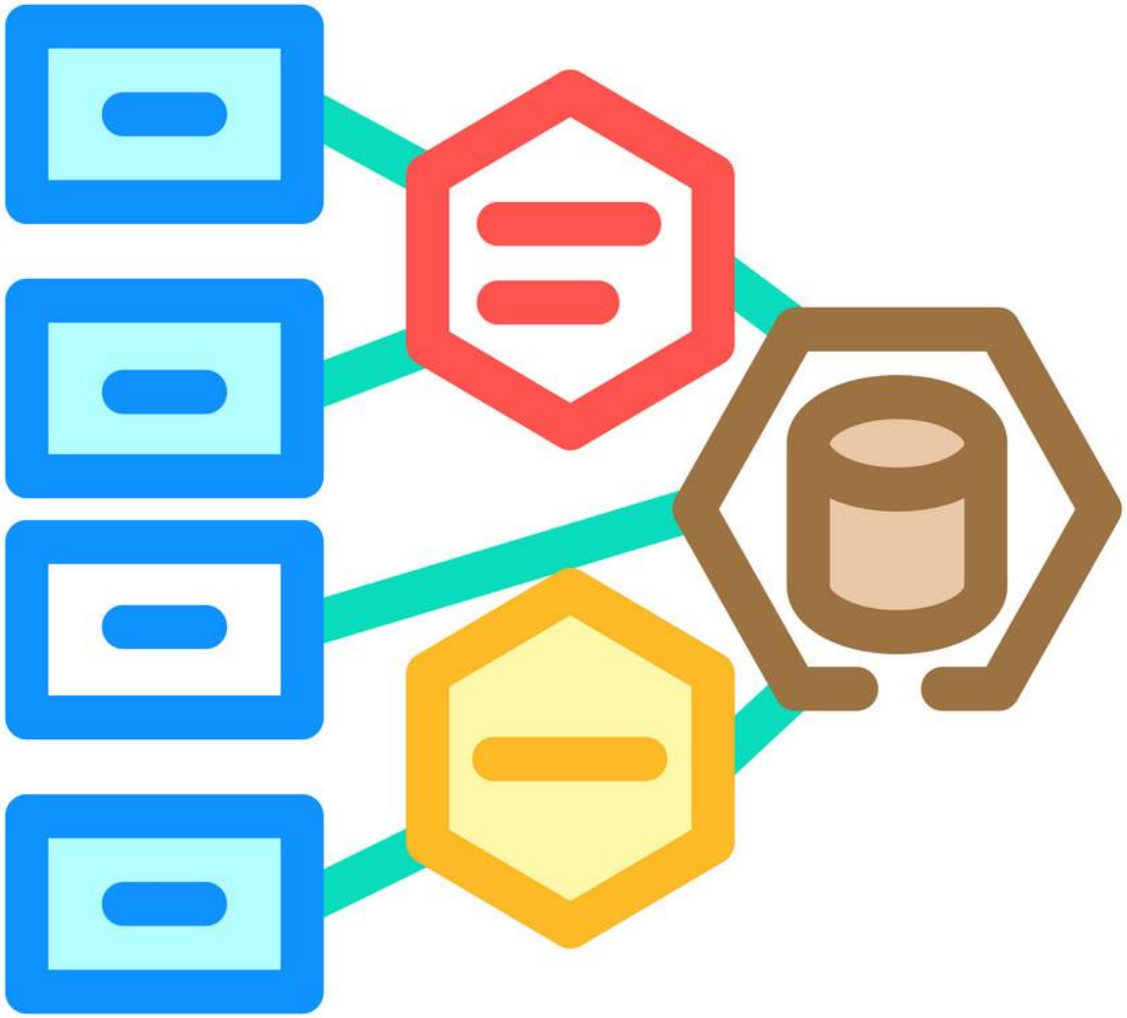
Software Design and architecture

Fall 2024

Dr. Natalia Chaudhry



Applications of Inheritance

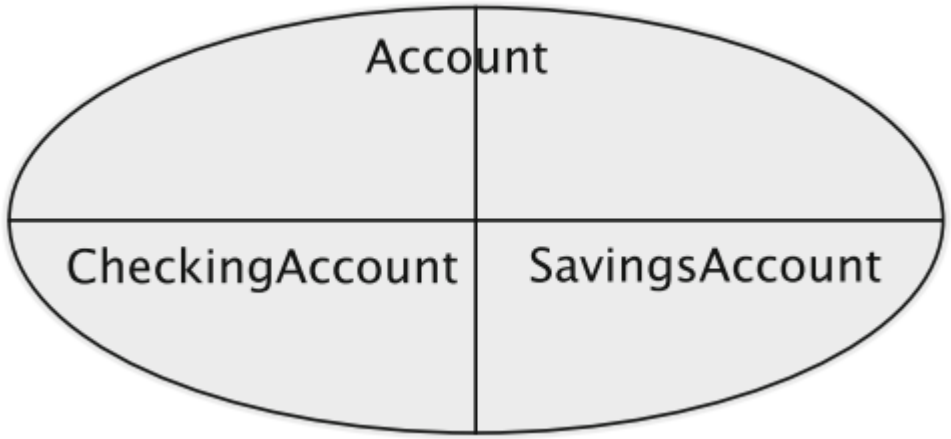
Restricting Behaviours and Properties

- One circumstance in which inheritance can be applied is when a class has characteristics that are a restriction of the characteristics of some other class.
- Suppose we have two classes, Rectangle and Square, to represent rectangles and squares. Every square is a rectangle in which length is equal to breadth, i.e., the property that length be equal to breadth restricts the number of rectangles that qualify to be classified as squares.
- Thus, Square is obtained from Rectangle by restricting a property; note that we are not attaching any more functionality to squares than rectangles.

- As a second example, suppose that we create a graphical user interface with many types of widgets, including labels.
- Suppose we have the requirement that the text in all labels be coloured blue.
- In this case, it is convenient to have a subclass that simply sets the colour to blue.

Abstract Superclass

- Sometimes the only purpose of having a superclass is to extract the common attributes and methods of potential subclasses, thus maximising reuse.
- No objects of the superclass itself are allowed, thus necessitating that the superclass be abstract.
- we have a set of subclasses that partition the universe of objects in the superclass



Adding Features

- we may extend an ancestor class, by adding new features, to get the descendant.
- ‘Moving Vehicle’ that can be defined by extending an existing ‘Vehicle’ and adding the attribute ‘speed.’

Hiding Features of the Superclass

- Sometimes we want to restrict behaviour by suppressing some functionality of the superclass.
- Such a kind of restriction is discussed in the following example, where some of the features are eliminated in the subclass.
- Let List be a class that allows the creation of a list in which objects can be added anywhere: in the front, at the tail, or at any position in-between.
- It is easy to get a class Queue that allows adding only at the tail and removing from the front.
- All other add and remove methods, which allow adding at or removing from other positions, should be disallowed.

- **Structural inheritance** is when compatibility between types is based on their structure (methods and properties) rather than an explicit relationship like subclassing or interface implementation.
- **Key Features:**
- No need for explicit extends or implements.
- Objects are interchangeable if they share the same structure.
- Common in dynamically-typed or structurally-typed languages (e.g., Python, Go).

```
class Bird:
    def fly(self):
        print("Flying!")

class Airplane:
    def fly(self):
        print("Gliding!")

def takeoff(flying_object):
    flying_object.fly()

takeoff(Bird())      # Works
takeoff(Airplane())  # Also works, no inheritance needed
```

Type inheritance

- where relationships are explicitly declared through class inheritance (class Derived : public Base) or
- interface implementation (via pure virtual functions).

Combining Structural and Type Inheritance

- We can also have situations where two kinds of inheritance are applied to define a class that suits our application.
- The most common of such situation is one where one superclass provides the necessary structure and another one, usually an interface, defines the function.
- A binary search tree, for instance, can be seen as a class that extends binary tree (structure inheritance) and implements the OrderedList interface (which defines the function of the binary search tree).
- The OrderedList operations are implemented using the methods provided by the class representing the binary tree, giving the name implementation inheritance to this usage

```
# Base class (Type Inheritance)
class Flyable:
    def fly(self):
        raise NotImplementedError("Subclass must
implement fly!")

# Type inheritance via subclassing
class Bird(Flyable):
    def fly(self):
        print("Bird is flying!")

# Structural inheritance (duck typing, no explicit
subclass)
class Airplane:
    def fly(self):
        print("Airplane is gliding!")

def takeoff(flying_object):
    flying_object.fly()  # Works as long as fly() exists
```

```
# Combining both
bird = Bird()          # Type inheritance
plane = Airplane()     # Structural inheritance

takeoff(bird)
takeoff(plane)
```

Inheritance: Some Limitations

- Subclassing could result in deep hierarchies, which usually makes it quite difficult to understand the code.
- In systems that do not support multiple inheritance, subclassing is not always feasible.
- It may be necessary to hide selected features of the superclass. For example, if we extend the class java.util.LinkedList to implement a queue, all of the methods of the superclass will be exposed, which may compromise integrity.
- Combining inheritance with genericity may result in complications due to implementation issues with a particular language.

That’s it