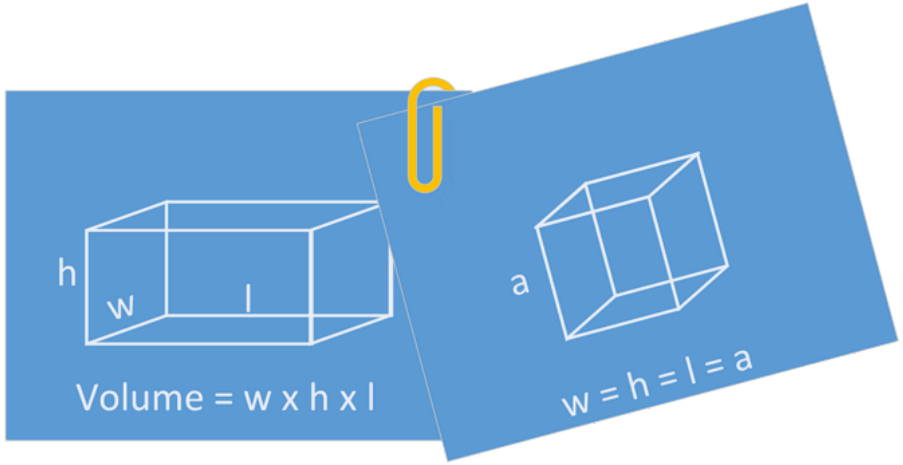
Inheritance

COSC346

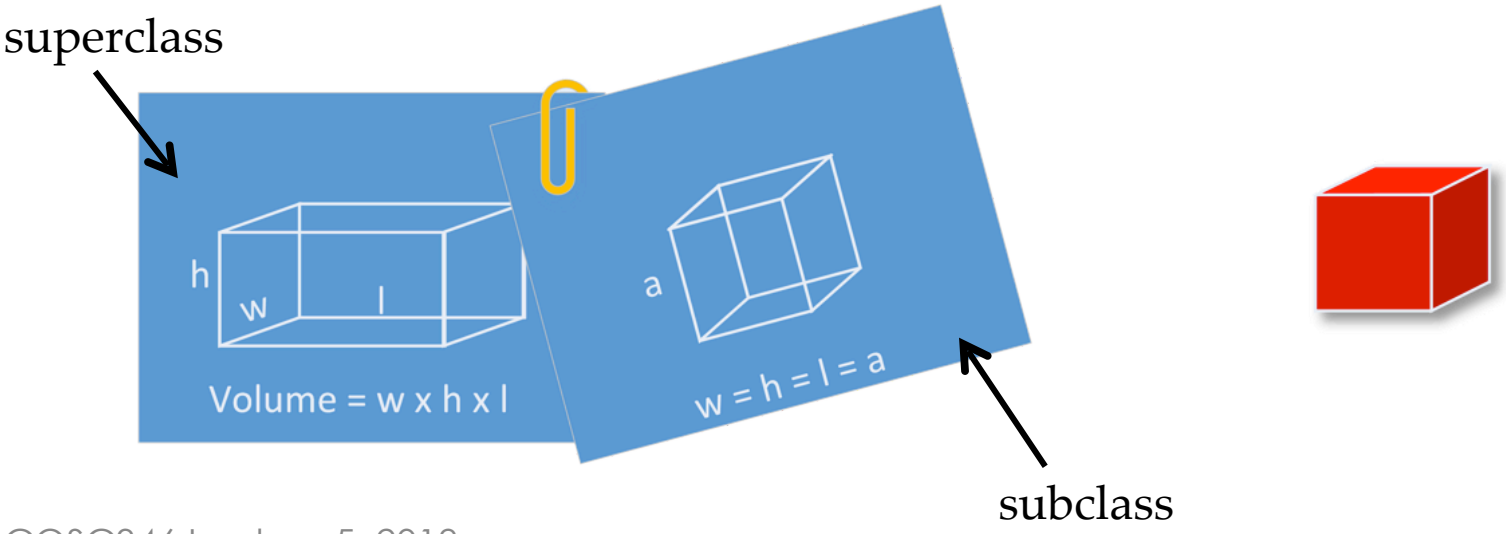
Inheritance

- Inheritance is the creation of a subclass from a previously existing class. It allows us to re-use code:
  - inheriting parent methods
  - adding new methods
  - modifying, or overriding, existing methods



Subclass and superclass

- Subclass extends its superclass
  - **Methods** are inherited by subclasses
  - **Member variables** are inherited by subclasses



Reasons for using inheritance

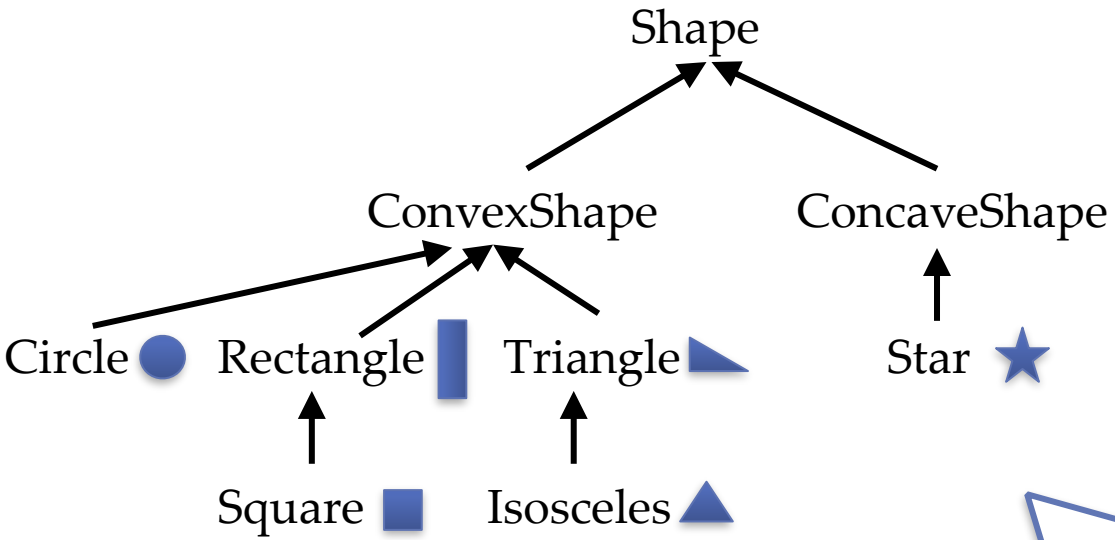
- **Specialisation**—subclass is a more specialised form of its parent
  - e.g., every square is a rectangle, not every rectangle is a square
- **Specification**—subclass implements behaviour described, but not implemented, by its parent
- **Extension**—subclass provides new behaviour and capabilities
- **Limitation**—subclass restricts behaviour of the parent class
- **Generalisation**—subclass modifies behaviour of the parent to create a more general kind of object

Is-a test

- Rule for testing whether two concepts should be linked by inheritance relationship
- If the sentence "Concept A **is a** concept B" sounds right, then inheritance is likely to be appropriate relationship
  - Is a rectangle a square?
  - Is a square a rectangle?
  - Is an integer a complex number?

Hierarchy

- Inheritance is transitive

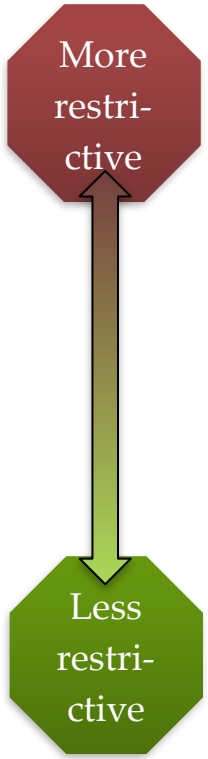


The arrows point to the parent like UML

Access Control and Inheritance

Generally in OOP access control affects what is visible from the derived classes:

- Inherited **private** methods and member variables
  - Not visible to the programmer writing the subclass code, nor the programmer using objects of the subclass
- Inherited **protected** methods and member variables
  - Visible to the programmer writing the subclass code, but not the programmer using objects of the subclass
- Inherited **public** methods and member variables
  - Visible to the programmer writing the subclass code as well as the programmer using objects of the subclass

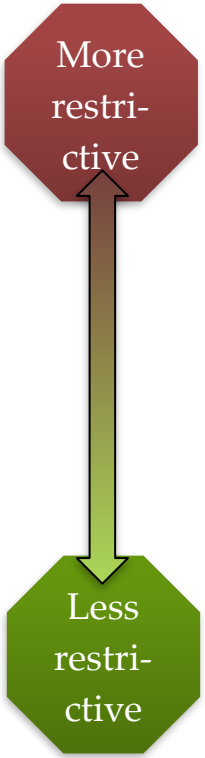


Access Control and Inheritance

Swift

In Swift rules of visibility have nothing to do with inheritance, and everything to do where the subclass is implemented:

- Inherited `private` methods and member variables are visible in the subclass only if it's implemented in the same file as the superclass.
- Inherited `fileprivate` methods and member variables are visible only in the defining source file.
- Inherited `internal` methods and member variables are visible in the subclass only if it's implemented in the same module as the superclass.
- Inherited `public` methods are always visible in the subclass regardless of where it's implemented.
- `Open` methods and classes can be subclassed anywhere and should be used sparingly.



Overriding methods

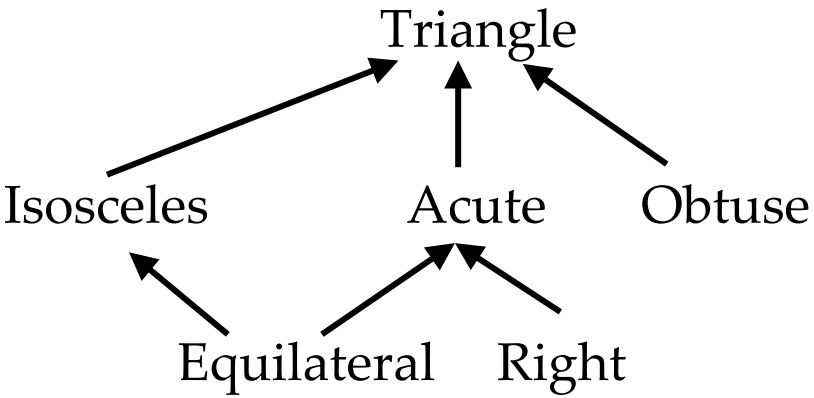
- A subclass can implement a method already defined/implemented by its superclass
  - In some languages (not Swift) parent methods cannot be overridden unless they have been declared as virtual
- The method from the lowest subclass in the hierarchy gets executed

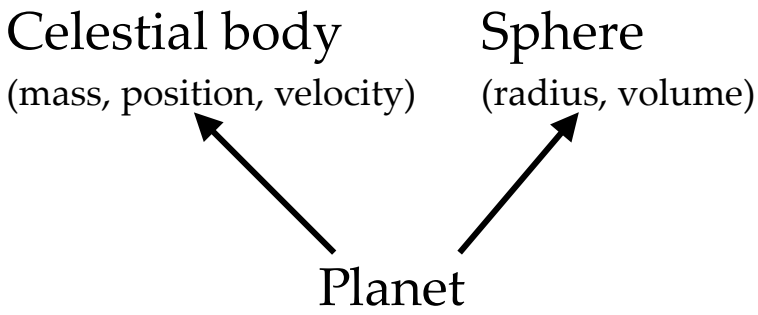
Override control

- **Final methods**—method in the superclass that cannot be overwritten
- **Abstract methods**—methods declared, but not implemented in the superclass: must be implemented in a subclass
  - Class that defines an abstract method is referred to as an **abstract class**—it cannot be instantiated, but it can be subclassed

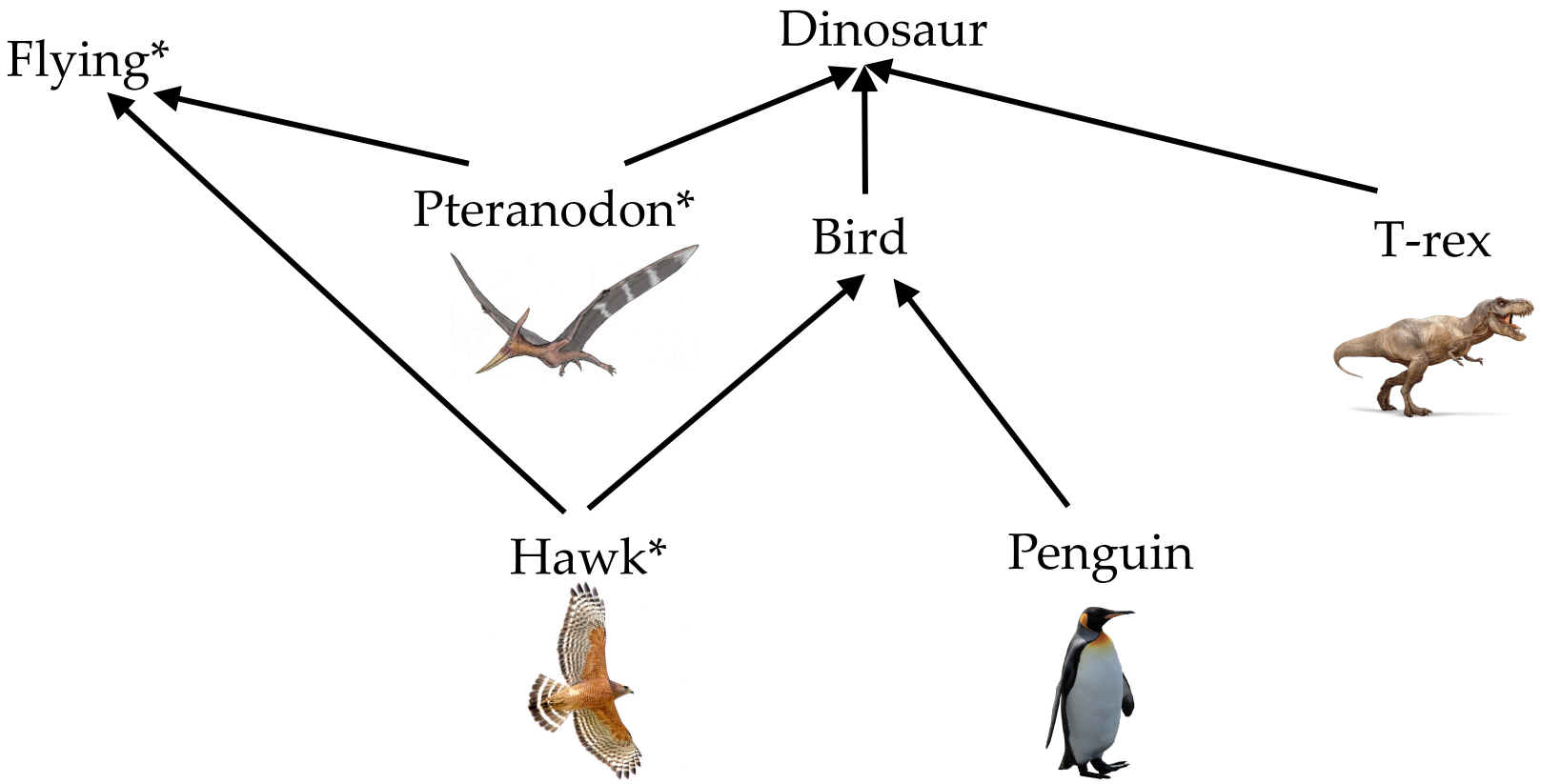
Multiple inheritance

- A scenario where a subclass has multiple parents



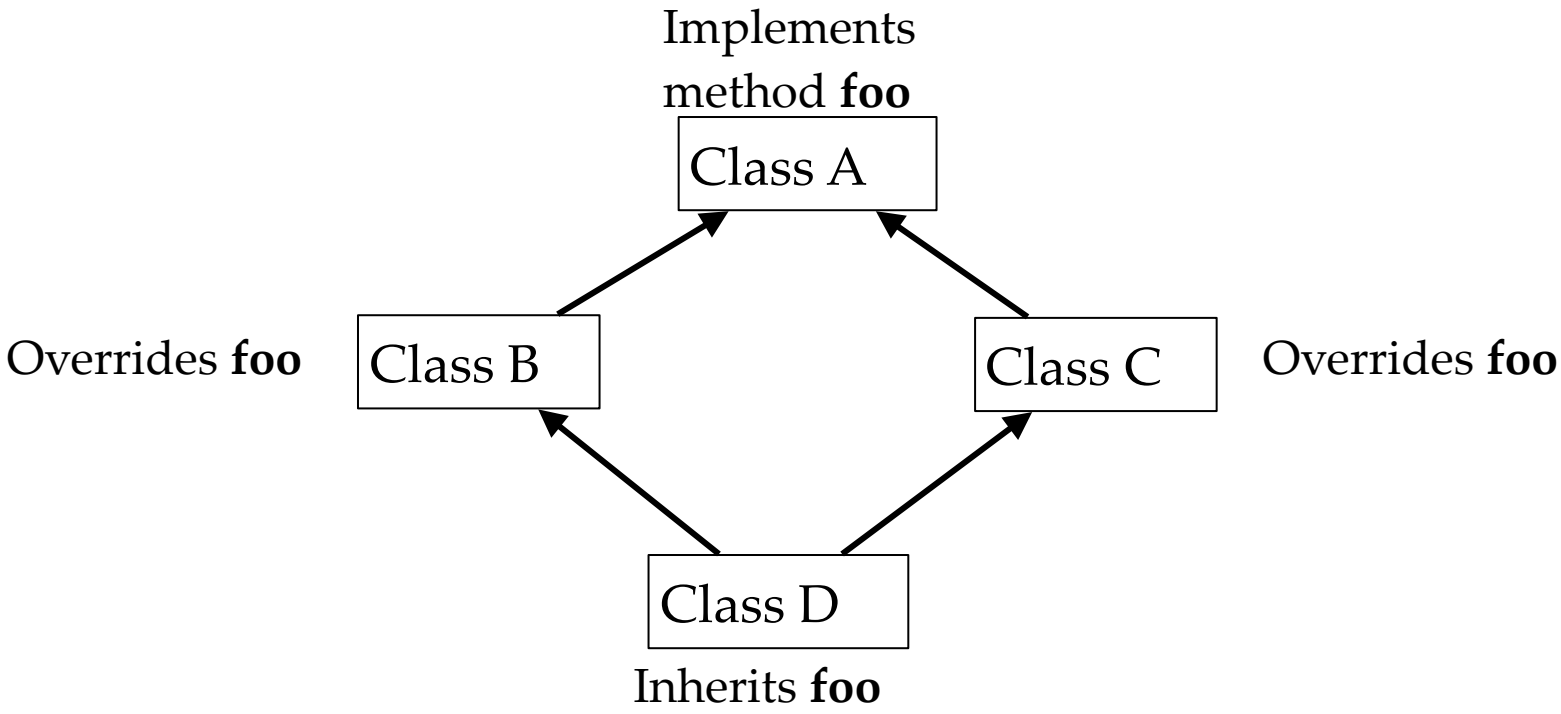


Multiple inheritance



Multiple inheritance

- Diamond of death

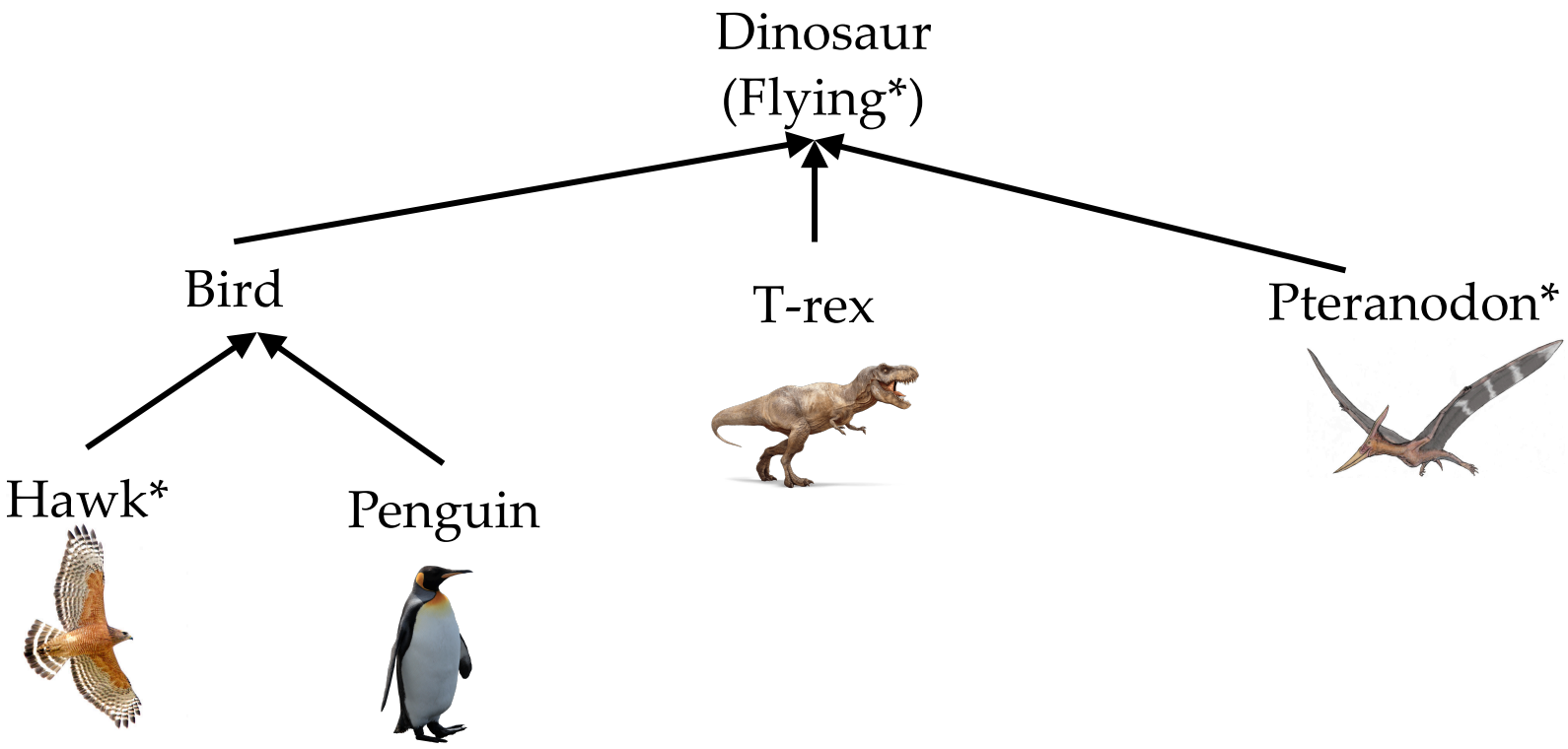


But which one? The one implemented in B or C?

Latent methods

- Latent methods - Richard O'Keefe
  - http://www.cs.otago.ac.nz/csis-seminars/pdfs/29-May-2015.pdf
  - Methods that are depend on a set of abstract methods
  - Subclass that implements the abstract methods that the latent method depends on, inherits the latent method
  - Subclass that does not implement the abstract methods that the latent method depends on, does not inherit the latent method

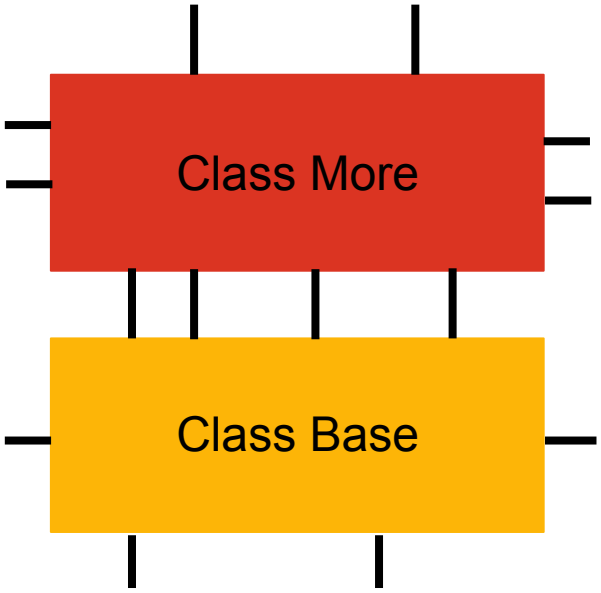
Latent methods



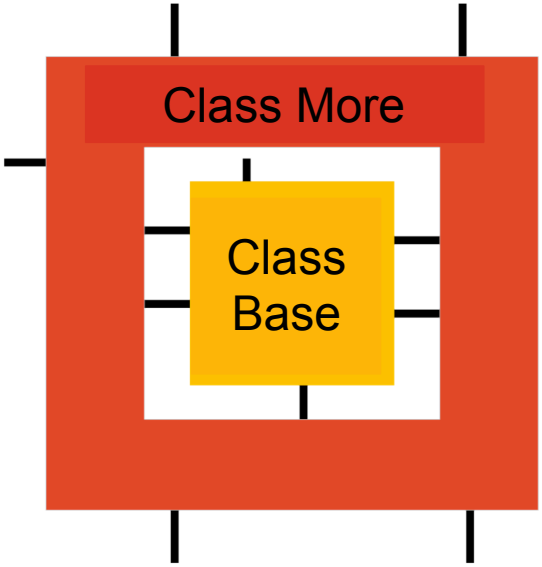
Composition

- Composition is where a class includes another class as its instance variable
- Has-a relationship
  - A rectangle has an edge
  - Address book has an entry for a person

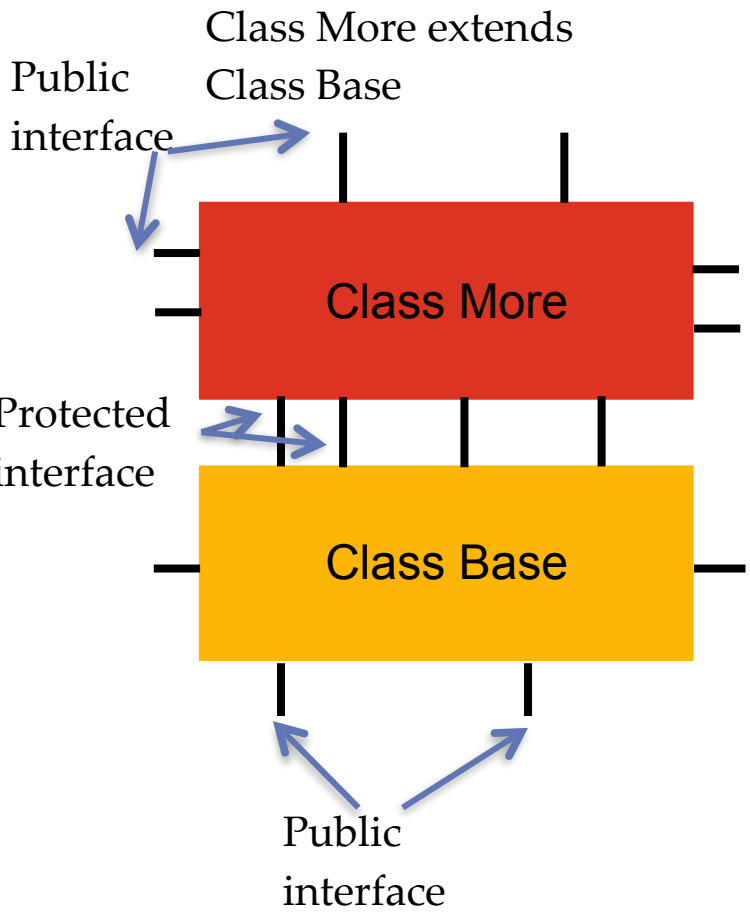
Inheritance versus composition



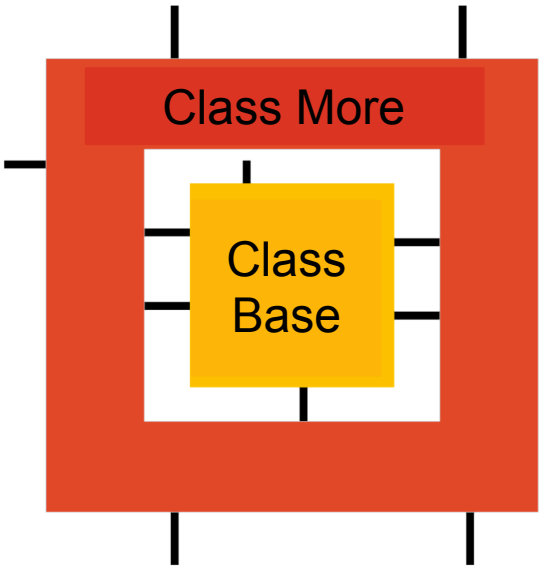
versus



Inheritance versus composition



versus



Design Pattern - Strategy

- Encapsulates each of a family of algorithms
- Allow the algorithm to vary independently from clients that use it
- Change algorithm at **runtime** in response to needs

- Different variants of an algorithm
  - Sorting algorithms with different space/time tradeoff.
- Related classes that differ only in behaviour
  - Different brake behaviours for Car class (with/without ABS)

Swift

Toolmaker

```swift
protocol PrintStrategy {
    func print(_ string: String) -> String
}

final class UpperCaseStrategy: PrintStrategy {
    func print(_ string: String) -> String {
        return string.uppercased()
    }
}

final class LowerCaseStrategy: PrintStrategy {
    func print(_ string:String) -> String {
        return string.lowercased()
    }
}
```

```swift
class Printer {

    private let strategy: PrintStrategy

    func print(_ string: String) -> String {
        return self.strategy.print(string)
    }

    init(strategy: PrintStrategy) {
        self.strategy = strategy
    }
}
```

Builder

```swift
var lower = Printer(strategy: LowerCaseStrategy())
var upper = Printer(strategy: UpperCaseStrategy())

print(lower.print("Hello, World!"))
print(upper.print("Hello, World!"))
```

Strategy in the real world?

Summary?