Inheritance (cont.)

COSC346

Benefits of inheritance

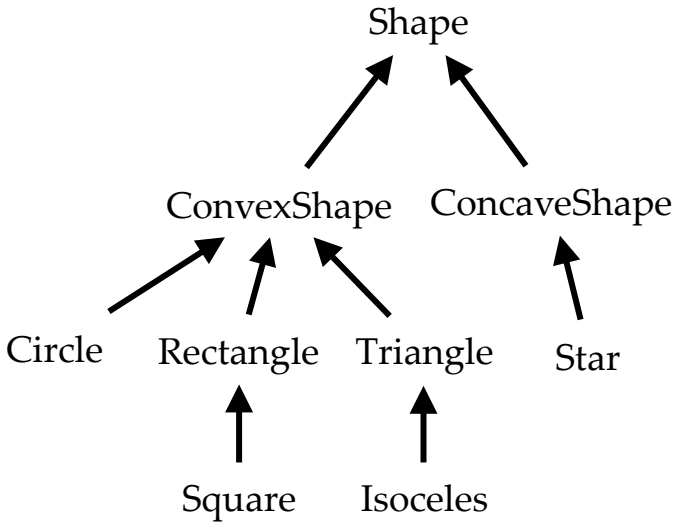
- **Code reusability**—no need to implement methods inherited from the parent
- **Interface consistency**—easy to conform to generic requirements for an interface while implementing only few methods
- **Code portability**—lower level routines (superclasses at the top of the tree) can be used in different projects

Cost of inheritance

- **Weakens encapsulation**—need to understand how the superclass works in order to use the subclass
- **Execution speed**—especially at initialisation time, when a series of constructors (initialisation routines) get invoked
- **Memory usage**—for a specialised subclass, is it worth carrying "extra baggage" of internal variables that comes with the parent class?

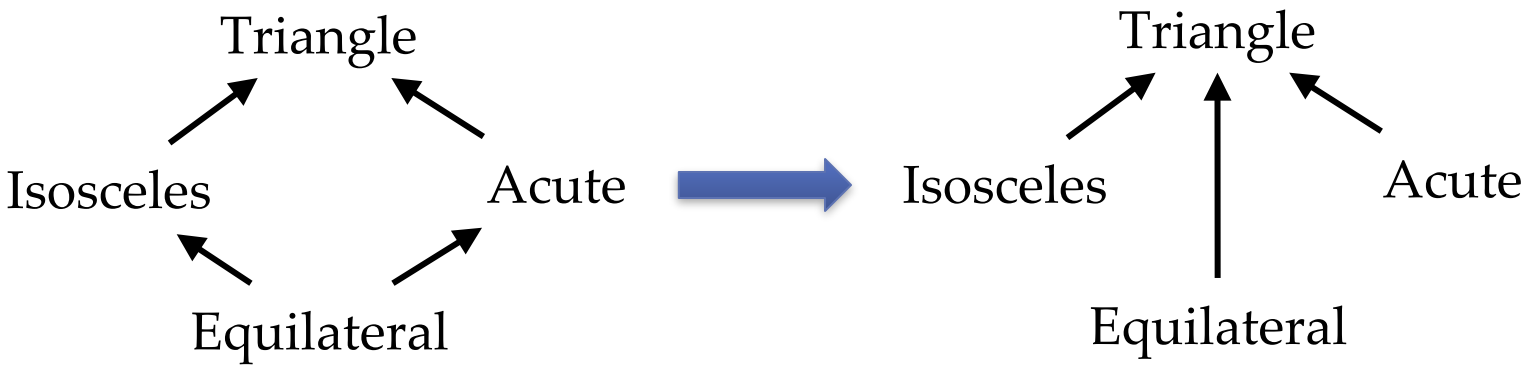
Upcasting and Downcasting

- Casting refers to treating objects as if they were of different types
- **Upcasting**—changing type label of object to that of its parent class
  - **Implicit cast**—never fails because child object **is a** parent object
  - Methods added by the child are not available after the upcast
  - Parent's methods that have been overridden by the child retain the overridden behaviour
- **Downcasting**—changing type label of an object to that of its child class
  - **Explicit cast**—can fail because a given object may or may not be an instance of the expected subclass
  - Usually done to reverse upcasting
  - Generally considered a bad practice



Inheritance in Swift

- In Swift a class can extend only one superclass—it can have only one parent
- Technically this makes Swift slightly less expressive
  - But it sidesteps the problems associated with multiple inheritance
  - Most of the time, a multiple inheritance hierarchy can be rearranged to conform to a single-parent paradigm



Example: Composition

Swift

```
public class XYPoint {
    public var x, y: Int

    public init(x: Int, y: Int) {
        self.x = x
        self.y = y
    }
}

public class Shape {
    internal var position: XYPoint

    public init(position: XYPoint) {
        self.position = position
    }

    public func translate(by translation: XYPoint) {
        self.position.x += translation.x
        self.position.y += translation.y
    }

}

var shape: Shape = Shape(position: XYPoint(x: 2, y: 3))

shape.translate(by: XYPoint(x: 3, y: 0))
```

Class Shape has a member variable that is an instance of class XYPoint

Inheritance

Class Rectangle inherits from class Shape

```
public class Rectangle : Shape {
    internal var w, h: Int

    public var area: Int {
        return self.w*self.h
    }

    public var description: String {
        return "Rectangle at (\(self.position.x),\(self.position.y))" +
               "of width: \(self.w) and height: \(self.h)"
    }

    public init(position: XYPoint, width: Int, height: Int) {
        self.w = width
        self.h = height
        super.init(position: position)
    }
}

var rect: Rectangle = Rectangle(position: XYPoint(x: 2, y: 3),
                                width: 4,
                                height: 2)

rect.translate(by: XYPoint(x: 3, y: 0))
print("Rectangle area is: \(rect.area)")
```

Class Rectangle defines new stored properties

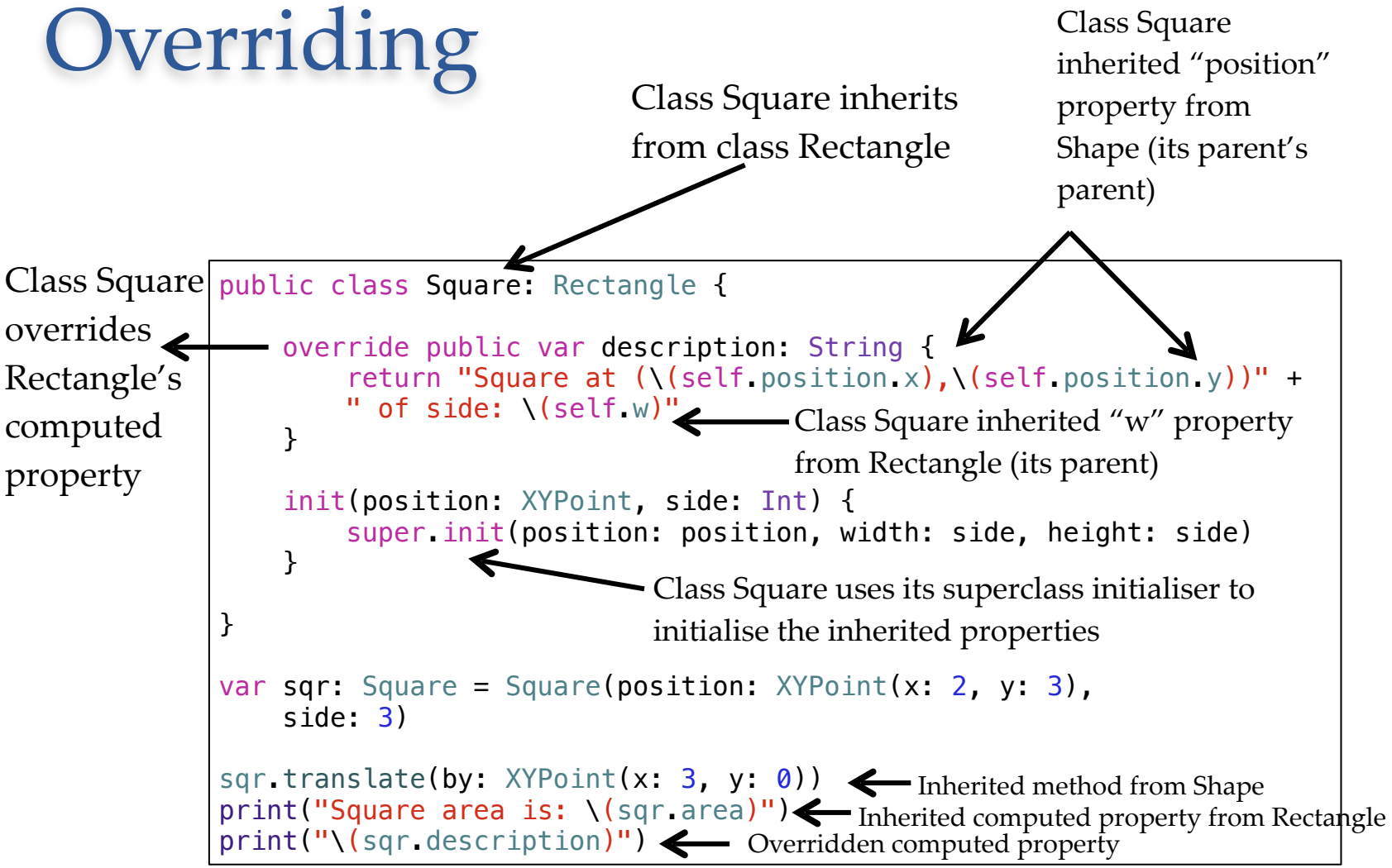
Class Rectangle defines a new computed properties

Class Rectangle inherited "position" property from Shape (its parent)

Class Rectangle uses its superclass initialiser to initialise the inherited properties

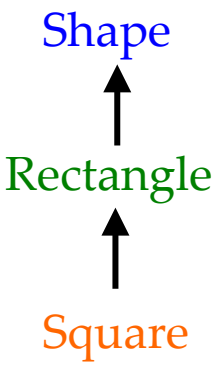
Class Rectangle also inherited Shape's "translate" method

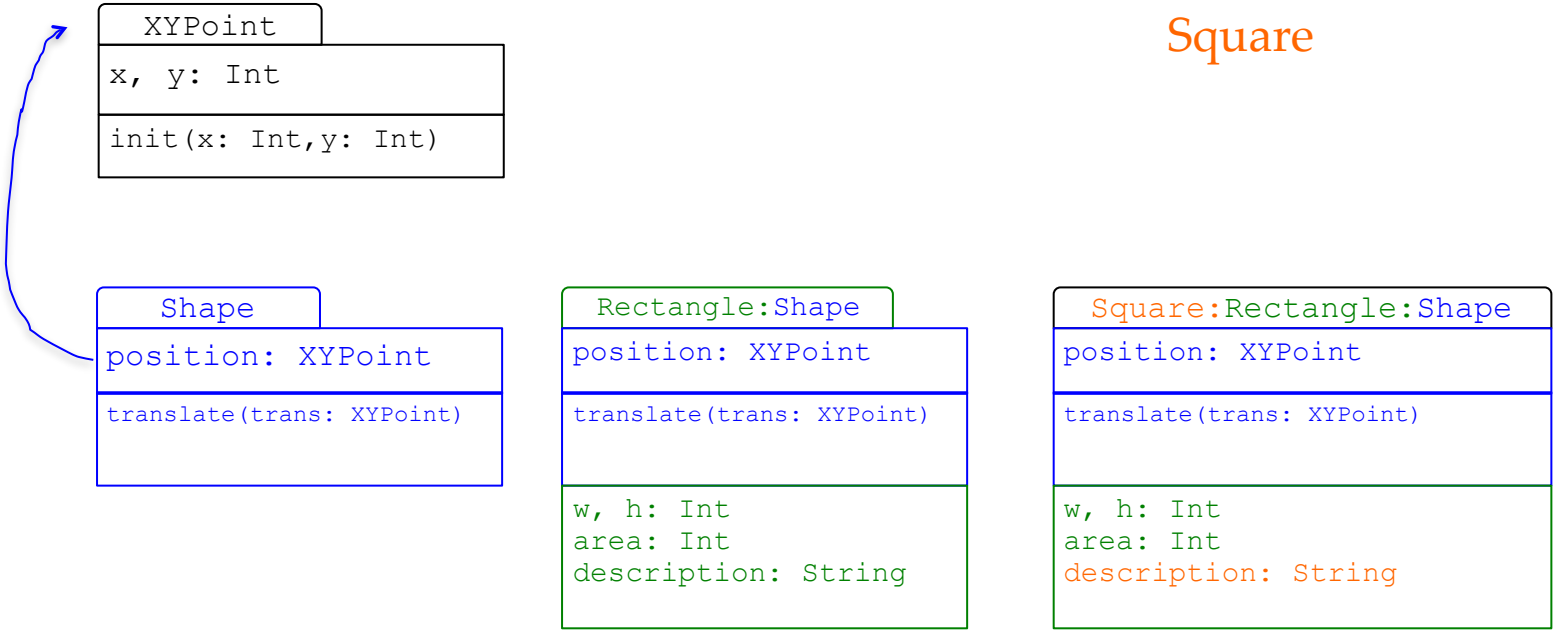
Overriding



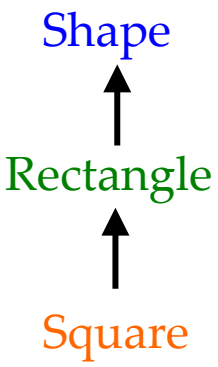
- **Note**: Swift doesn't have abstract methods/classes, but similar behaviour can be achieved with protocols

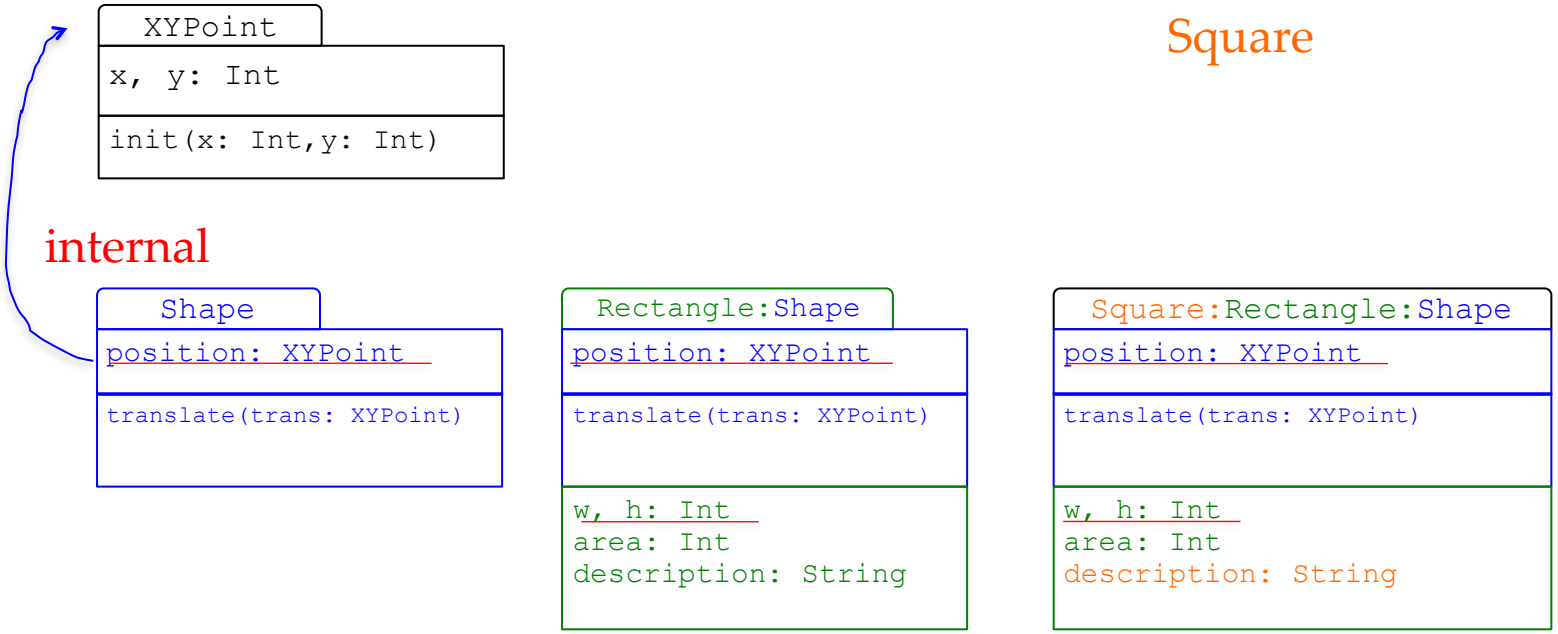
Example: Class hierarchy





Example: Access control (public)





Upcasting

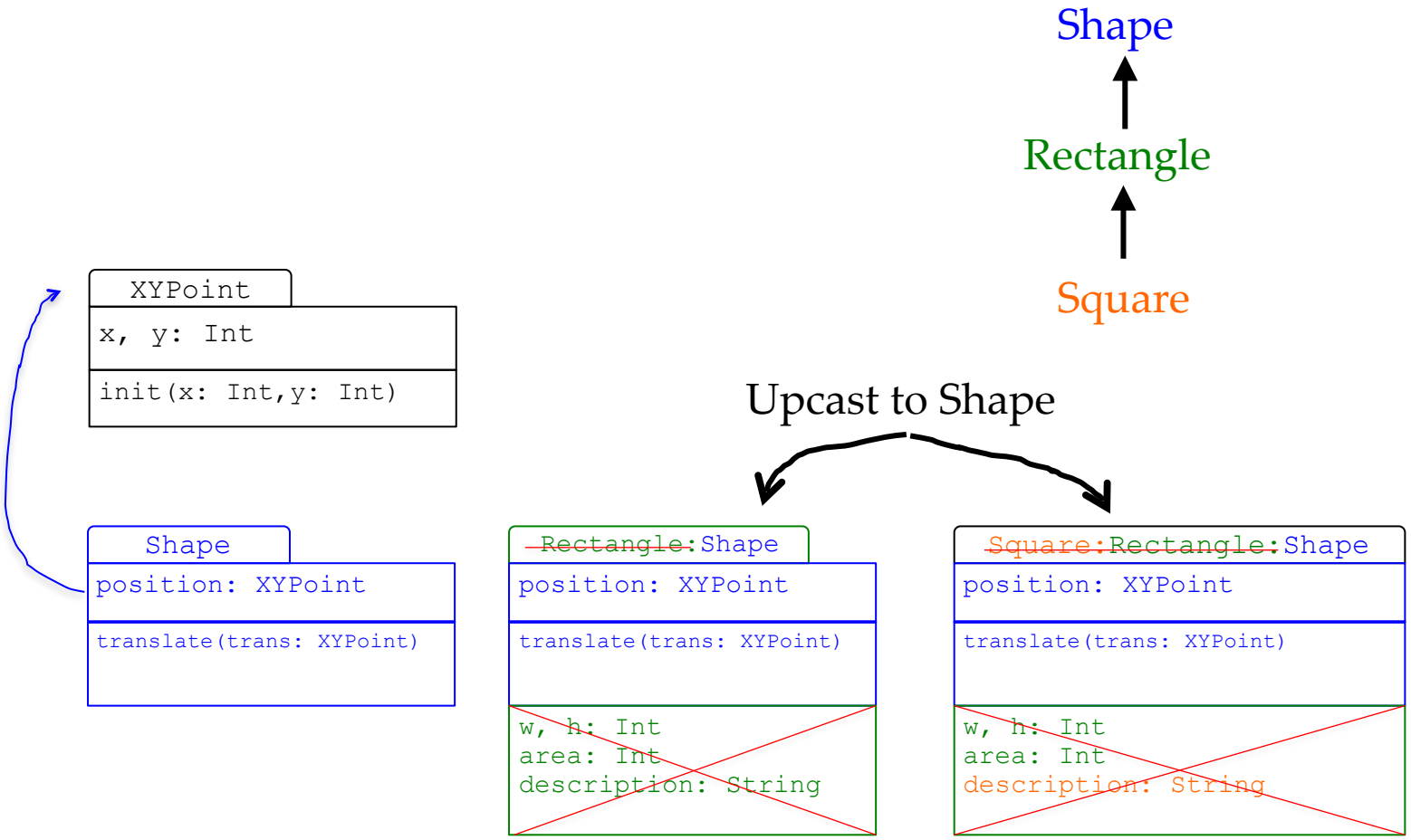
Accepts argument of type Shape

```
func willTranslate(shape: Shape, by translation: XYPoint) {
    shape.translate(by: translation)
}

var shape: Shape = Shape(position: XYPoint(x: 2, y: 3))

var rect: Rectangle = Rectangle(position: XYPoint(x: 2, y: 3),
                                width: 4,
                                height: 2)

var sqr: Square = Square(position: XYPoint(x: 2, y: 3),
    side: 3)

willTranslate(shape: shape, by: XYPoint(x: -1, y: 4))

willTranslate(shape: rect, by: XYPoint(x: -2, y: -2))
willTranslate(shape: sqr, by: XYPoint(x: 2, y: 0))
```

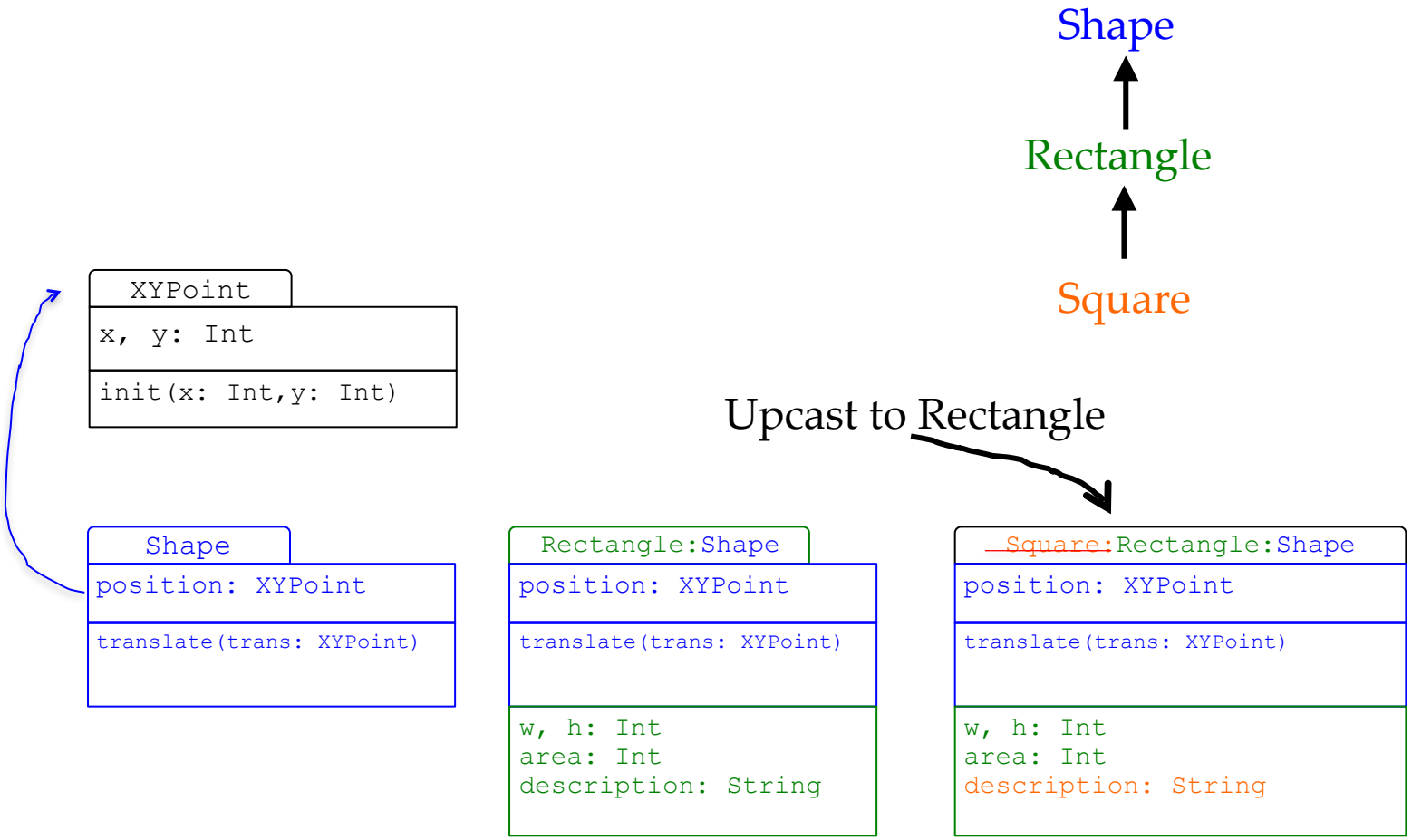
Rectangle is a Shape, so it's automatically upcasted

Square is a Shape, so it's automatically upcasted

Example: Upcasting



Example: Upcasting



Note that the upcast object will still execute the overridden (and not the parent's) method

Downcasting

Swift

```
func mightGiveArea(shape: Shape) {

    if shape is Rectangle {
        let rect = shape as! Rectangle
        print("Area is \(rect.area)")
    }
}

var shape: Shape = Shape(position: XYPoint(x: 2, y: 3))
var rect: Rectangle = Rectangle(position: XYPoint(x: 2, y: 3),
                                width: 4,
                                height: 2)
var sqr: Square = Square(position: XYPoint(x: 2, y: 3),
                         side: 3)

mightGiveArea(shape)
mightGiveArea(rect)
mightGiveArea(sqr)
```

Forced downcast of Shape to Rectangle

Prints nothing

Prints area

Prints area

Downcasting

Swift

```
func mightGiveArea(shape: Shape) {

    if let rect = shape as? Rectangle {
        print("Area is \(rect.area)")
    }
}

var shape: Shape = Shape(position: XYPoint(x: 2, y: 3))
var rect: Rectangle = Rectangle(position: XYPoint(x: 2, y: 3),
                                width: 4,
                                height: 2)
var sqr: Square = Square(position: XYPoint(x: 2, y: 3),
                         side: 3)

mightGiveArea(shape)
mightGiveArea(rect)
mightGiveArea(sqr)
```

Prints nothing

Prints area

Prints area

Extension

- Swift supports class extensions, where you can add methods and properties to existing class
- Don't need the source code of the original class in order for the extension to work

```
extension Rectangle {
    func perimeter() -> Int {
        return self.w*2+self.h*2
    }
}

var rect: Rectangle = Rectangle(position: XYPoint(x: 2, y: 3),
                                width: 4,
                                height: 2)
var sqr: Square = Square(position: XYPoint(x: 2, y: 3),
                         side: 3)

print("Rectangle perimeter is: \(rect.perimeter())")
print("Square perimeter is: \(sqr.perimeter())")
```

Add a new method to Class Rectangle

Use the new method on a Rectangle object

Class Square automatically inherits the extension

Design Pattern - Facade

- Provides a simple interface to a more complex subsystem
  - subsystem components can still be accessed
  - can add functionality and not just 'pass through'
- Principle of Least Knowledge
  - "talk only to your immediate friends"
    - self, parameters, instances, components

Swift

Toolmaker

```
class CPU{
    func freeze() {}
    func jump(position: Int) {}
    func execute(){}
}

class Memory{
    func load(position: Int, data: Int8[]){}
}

class HardDrive{
    func read(position: Int, size: Int) -> Int8[]{
        return [Int8]
    }
}
```

```
class Computer {
    let processor: CPU
    let ram: Memory
    let hd: HardDrive

    init(){
        processor = CPU()
        ram = Memory()
        hd = HardDrive()
    }

    func start(){
        processor.freeze()
        ram.load(BOOT_ADDRESS,
           hd.read(BOOT_SECTOR, SECTOR_SIZE))
        processor.jump(BOOT_ADDRESS)
        processor.execute()
    }
}
```

Builder

```
let computer = Computer()
computer.start()
```

Facade in the real world?

Summary?