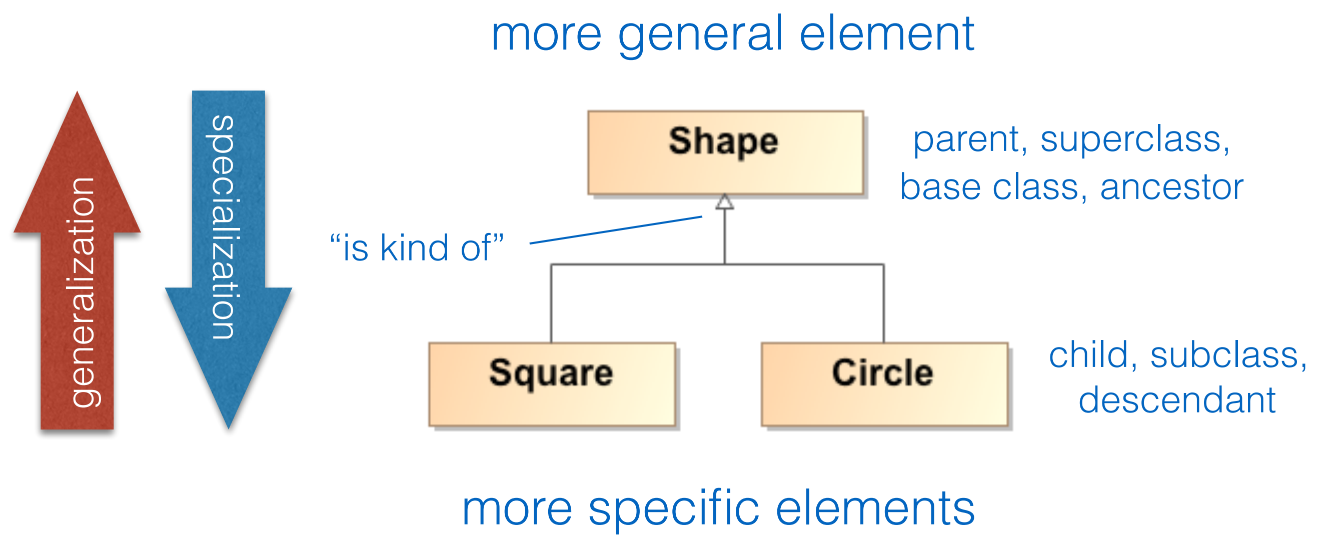
Analysis - inheritance & polymorphism

Generalization

- A relationship between a more general element and a more specific element
- The more specific element is entirely consistent with the more general element but contains more information
- The substitutability principle - an instance of the more specific element may be used where an instance of the more general element is expected

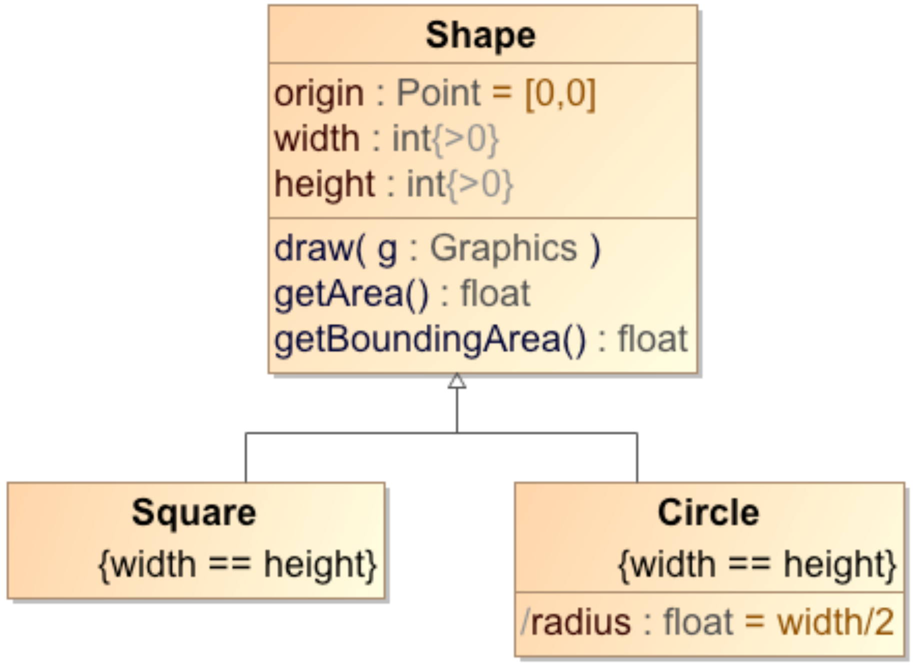
Class generalization



- A simple generalization hierarchy
- It is important to learn the terminology!

Class inheritance

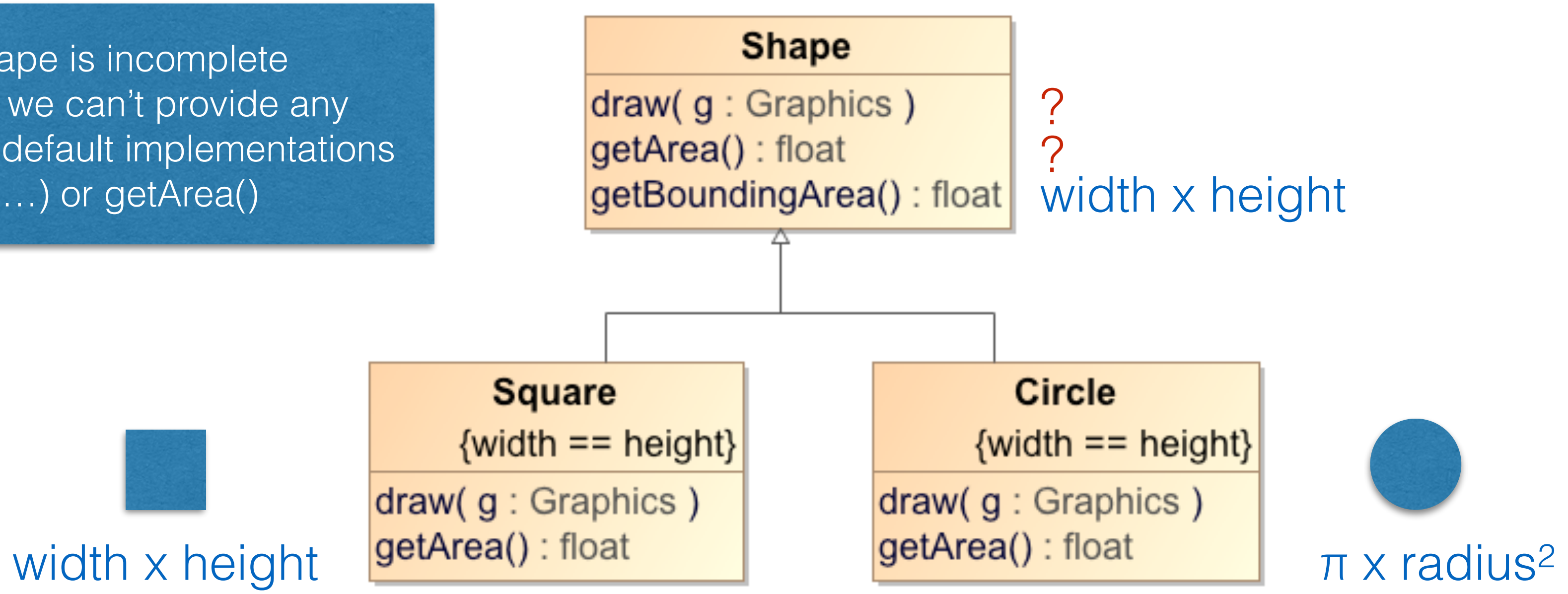
- Subclasses inherit all features of their superclasses: attributes, operations, relationships, stereotypes, tags, constraints etc.
- Subclasses can add new features
- Subclasses often need to override superclass operations
  - Note: it is *impossible* to override attributes
- Substitutability principle: We can use a subclass instance *anywhere* a superclass instance is expected



What’s wrong with these subclasses???

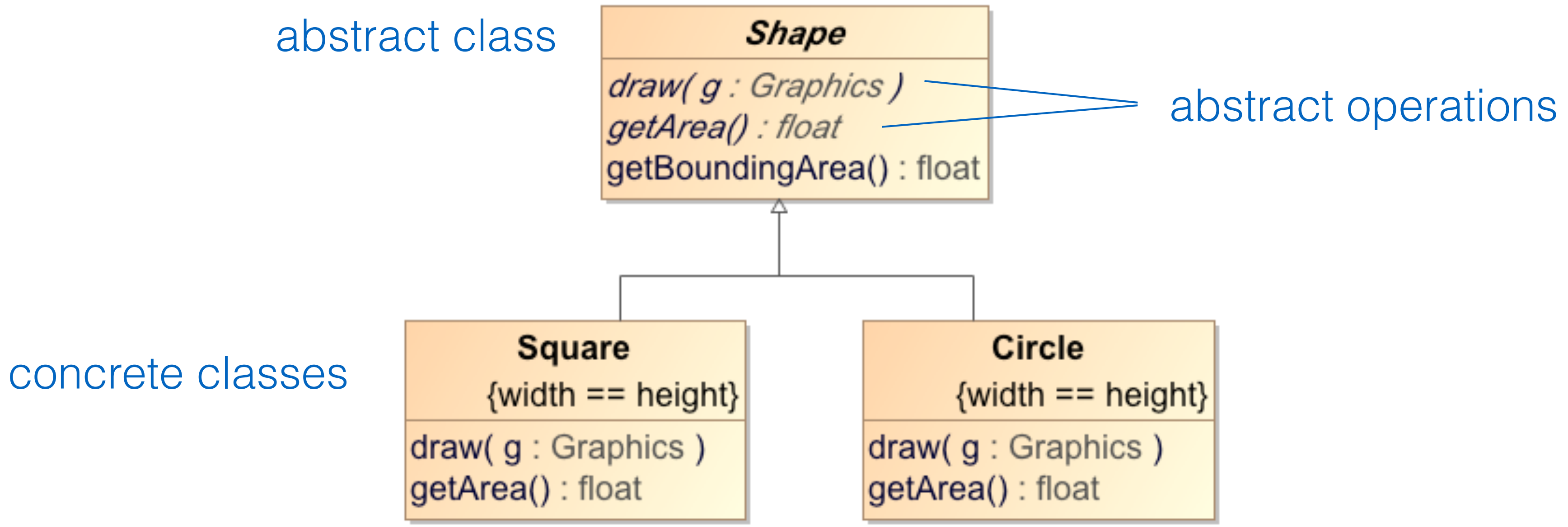
Overriding

Note: Shape is incomplete because we can't provide any sensible default implementations for draw(…) or getArea()



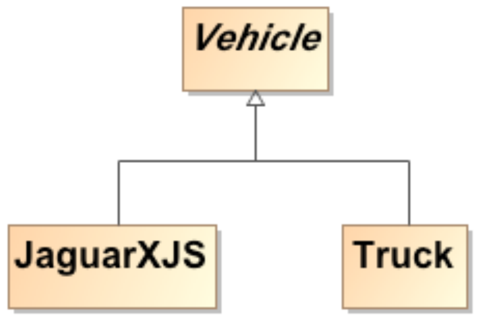
- Subclasses often need to override superclass behavior
- To override a superclass operation, a subclass must provide an operation with the same operation signature comprising operation name, return type and types of all the parameters. Parameter names don't count as part of the signature
- You can override abstract and concrete operations, but overriding concrete operations may be dangerous and is considered to be bad style!

Abstract classes & operations



- We can't provide an implementation for *Shape :: draw(g : Graphics)* or for *Shape :: getArea() : float* because we don't know how to draw or calculate the area for a "shape"!
- Operations that lack an implementation are *abstract operations*
- A class with any abstract operations can't be instantiated and is therefore an *abstract class*

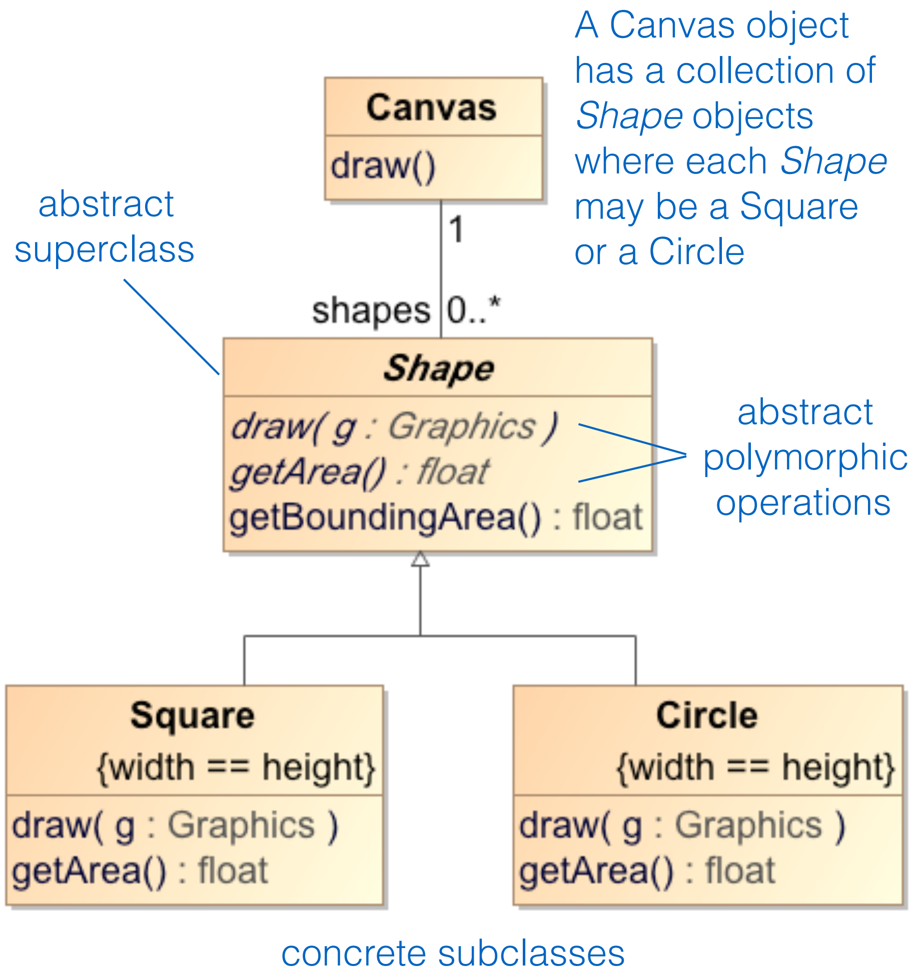
Exercise



- What’s wrong with this model?

Polymorphism

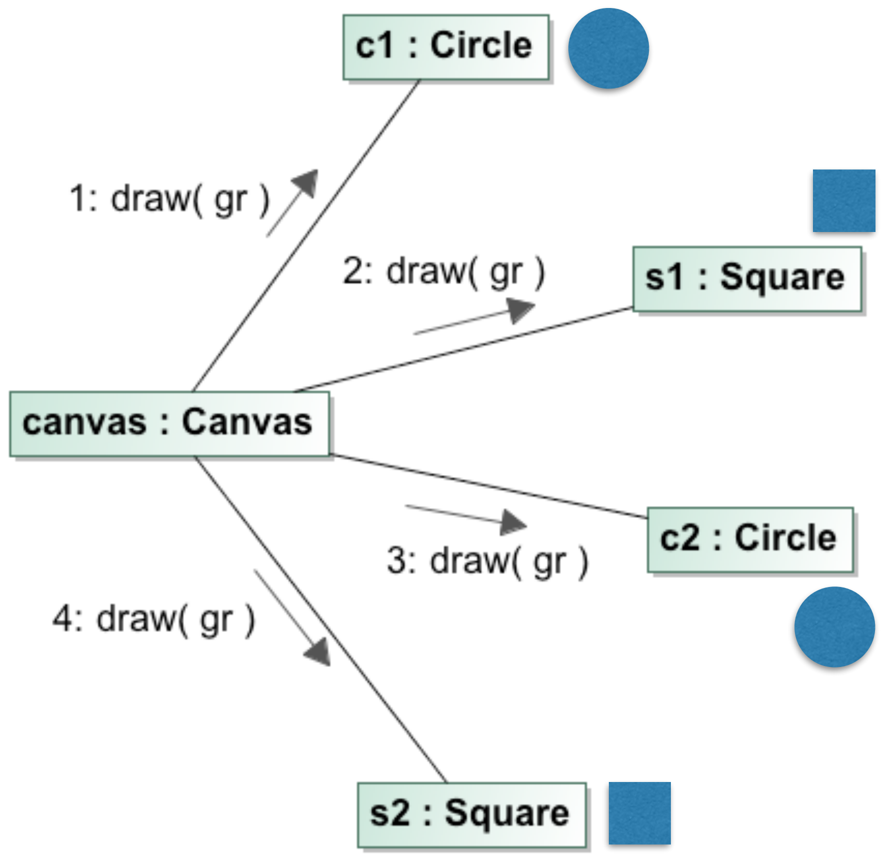
- Polymorphism = "many forms"
- A polymorphic operation has many implementations
- *Shape::draw(…)* and *Shape::getArea()* are polymorphic operations because Square and Circle both provide implementations for them
- All concrete subclasses of *Shape must* provide concrete draw(…) and getArea() operations because they are abstract in the superclass
- For draw(…) and getArea() we can treat all subclasses of *Shape* in a similar way - we have defined a *contract* for *Shape* subclasses



Shape abstract operations define a contract for all subclasses

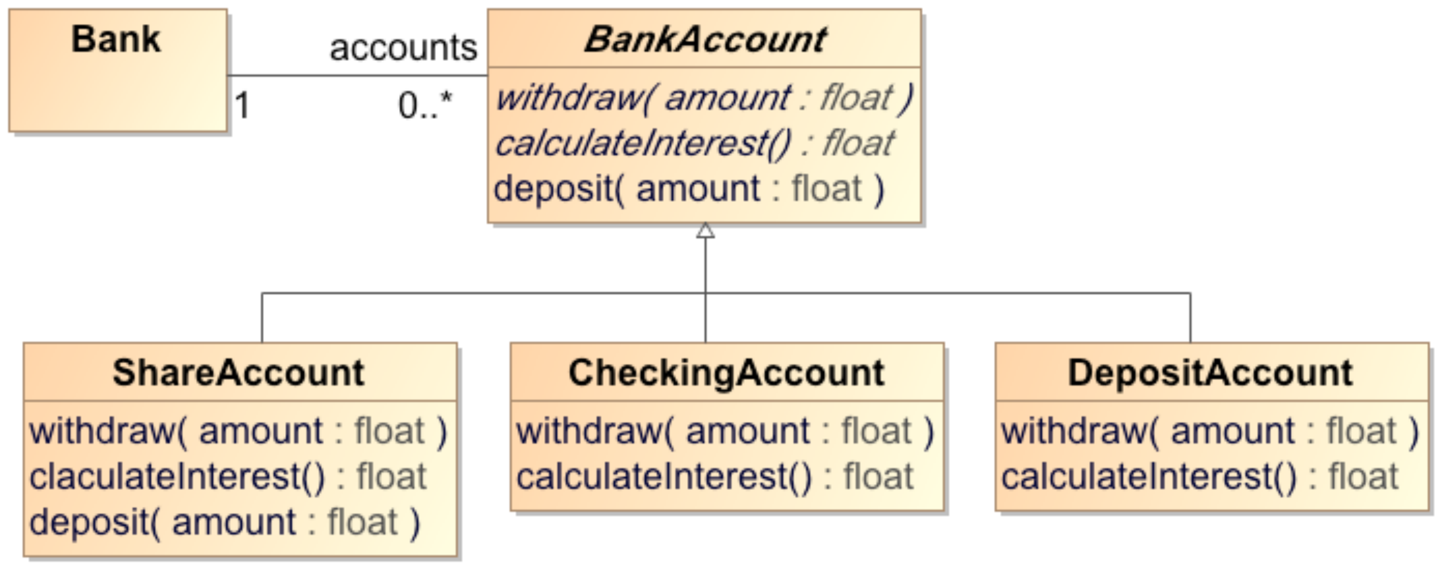
Polymorphism in action

- Each class of object has its own implementation of the draw(…) operation
- On receipt of the draw(…) message, each object invokes the draw(…) operation specified by its class
- We can say that each object "decides" how to interpret the draw(…) message based on its class



Note: this is a communication diagram - see later

BankAccount example


Bank
accounts
1
0..*
BankAccount
withdraw(amount : float)
calculateInterest() : float
deposit(amount : float)
ShareAccount
withdraw(amount : float)
claculateInterest() : float
deposit(amount : float)
CheckingAccount
withdraw(amount : float)
calculateInterest() : float
DepositAccount
withdraw(amount : float)
calculateInterest() : float

- In ShareAccount, we have overridden the deposit(…) operation even though it is not abstract. This is perfectly legal, and quite common, although it is generally considered to be bad style and should be avoided if possible

Summary

- Substitutability principle: We can use a subclass instance *anywhere* a superclass instance is expected
- Subclasses:
    - Inherit all features from their parents including constraints and relationships
    - May add new features, constraints and relationships
    - May override superclass operations
- A class that can’t be instantiated is an *abstract* class