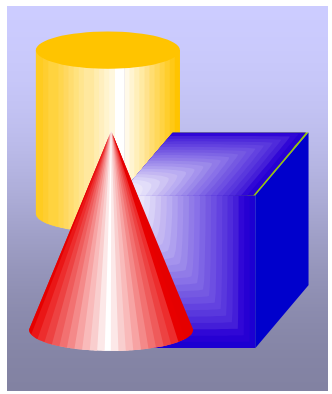
SmartImpart™

# Object Orientation

## Unit 2

## Object Oriented Programming Concepts



# Object Oriented Programming Concepts

- Topics
  - OOP
  - Structure & Behavior of a Class
  - State, Behavior & Identity
  - Abstraction
  - Encapsulation
  - Hierarchies
  - Strong Typing
  - Interface
  - Inheritance
  - Polymorphism
  - Delegation

# Procedural vs. Object-Oriented Programming

- The unit in procedural programming is *function*, and unit in object-oriented programming is *class*
- Procedural programming concentrates on creating functions, while object-oriented programming starts from isolating the classes, and then look for the methods inside them.
- Procedural programming separates the data of the program from the operations that manipulate the data, while object-oriented programming focus on both of them

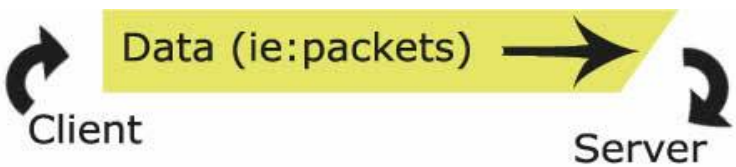



figure1: procedural

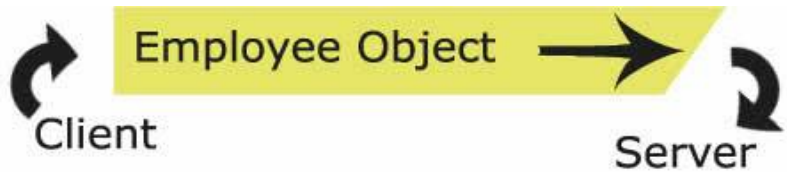
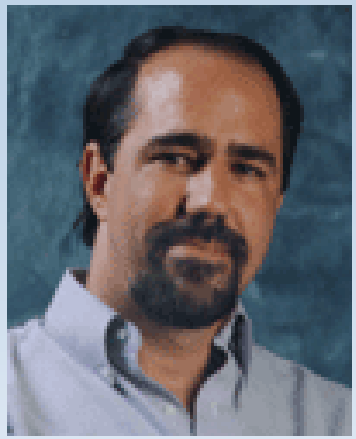



figure2: object-oriented

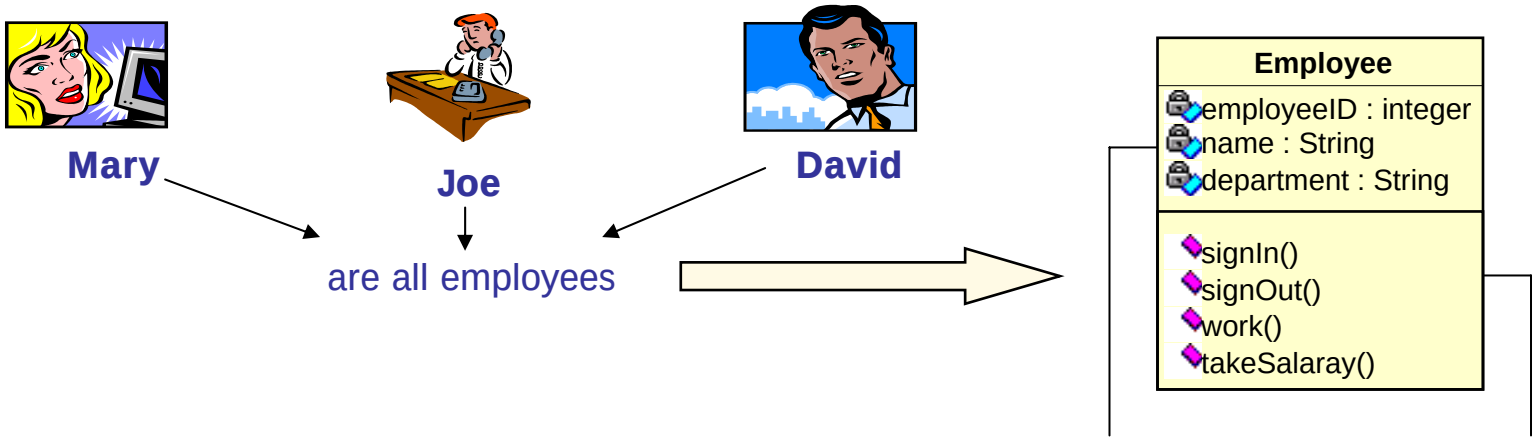
# Object-Oriented Programming

*“Object-oriented programming is a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class...”*



**Grady Booch**

# Structure & Behavior




**Classes are set of objects that share a common structure and behavior**

- The structure of a class includes its
  - Primary structure
  - Reused structure
- The behavior of a class includes its
  - Visible behavior
  - Implementation behavior

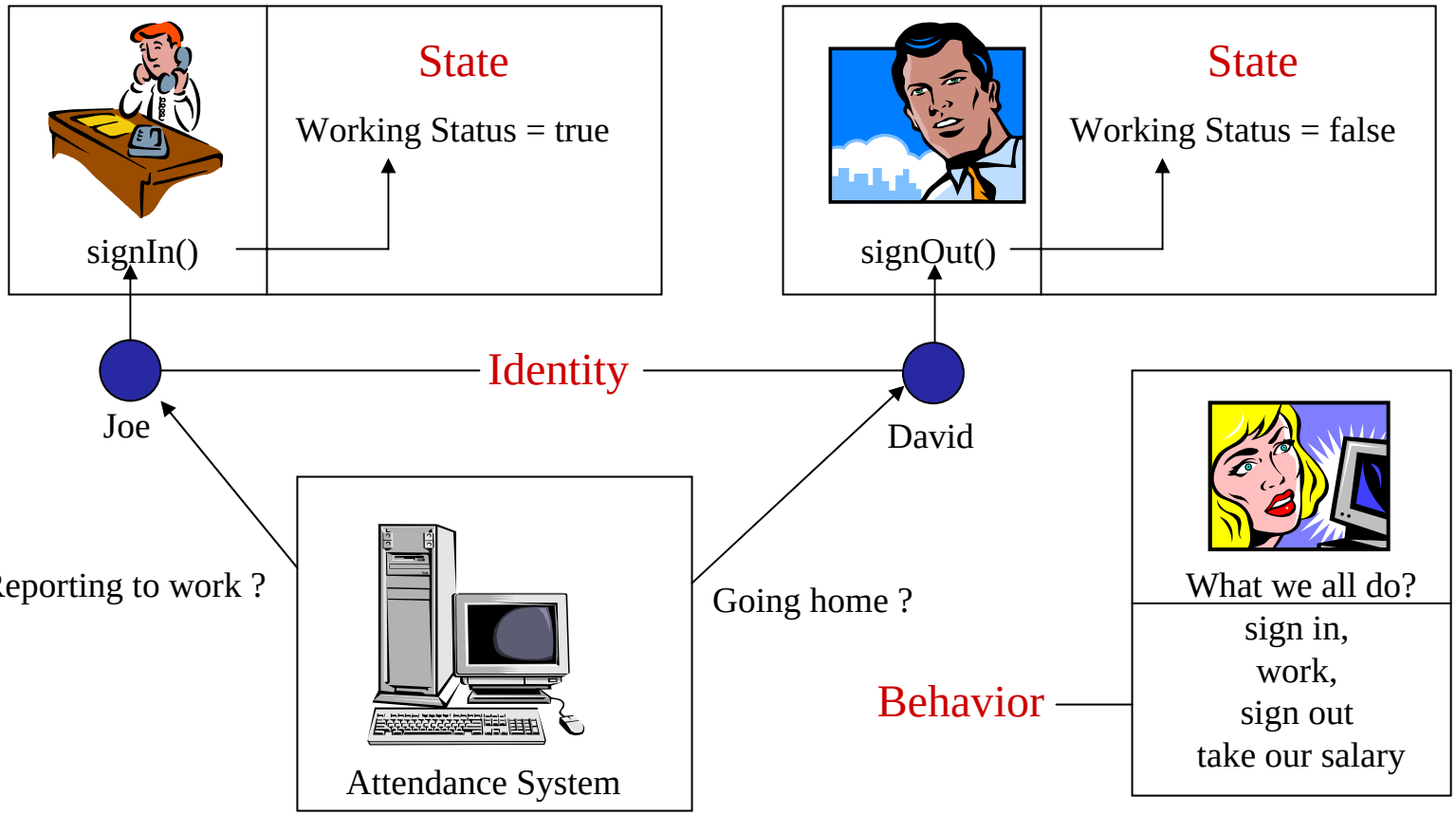
# State, Behavior and Identity

- An object is an entity with a well-defined
  - State
  - Behavior
  - Identity

- **State**
  - The set of values that an object holds
- **Behavior**
  - How an object acts and reacts, in terms of its state changes and message passing
  - The visible behavior of an object is modeled by the set of messages it can respond to
- **Identity**
  - That property of an object which distinguishes it from all other objects

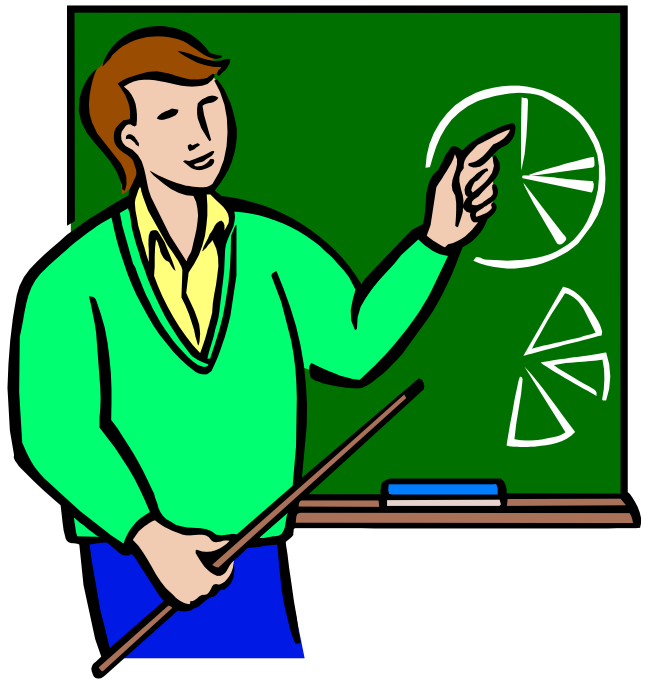
# State, Behavior and Identity

Understanding simplified !!!



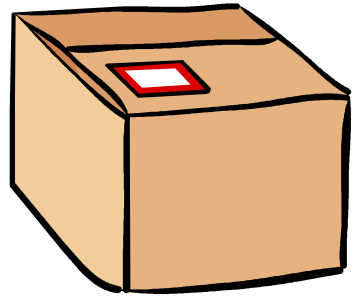
# Basic Object-oriented Principles

- Abstraction
- Encapsulation
- Hierarchies



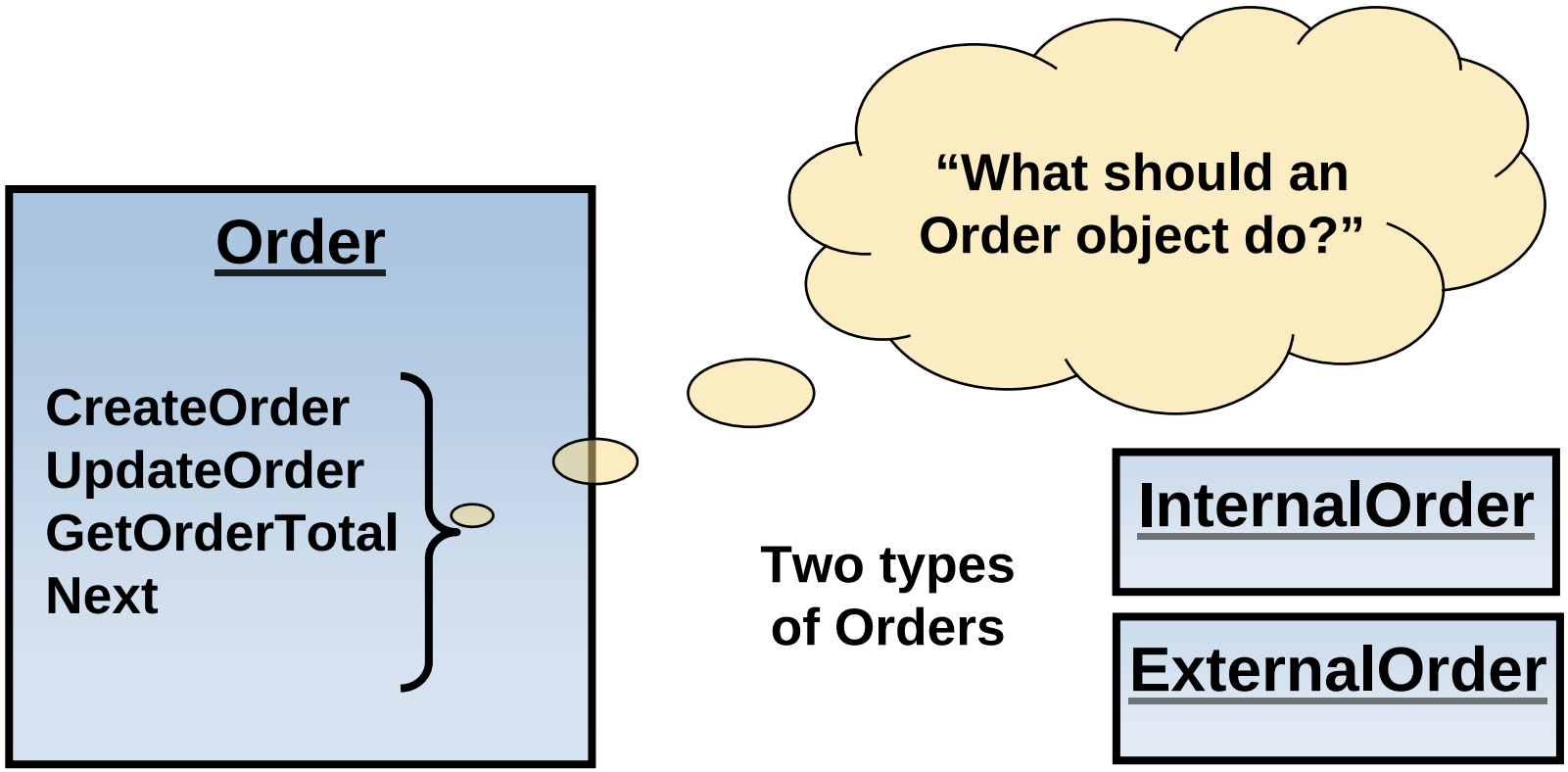
# Abstraction

***Public View of an Object***



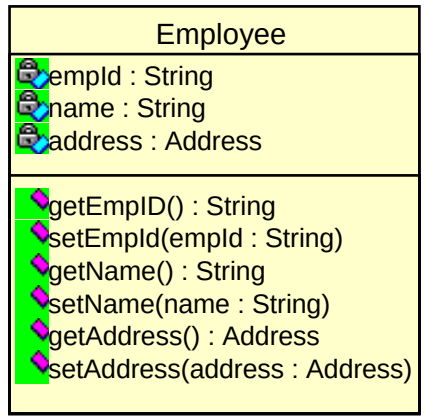
- *Abstraction* is used to manage complexity
  - Focus on the essential characteristics
  - Eliminate the details
  - Find commonalities among objects
- Defines the public contract
  - Public definition for users of the object
  - The “Outside view”
  - Independent of implementation

# Abstraction - Example

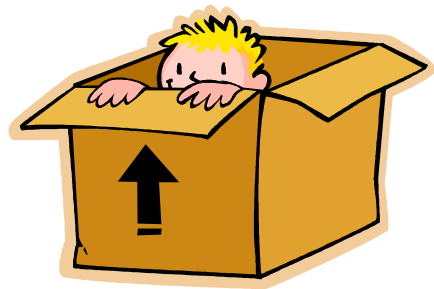


# Encapsulation

## *Hide Implementation Details*

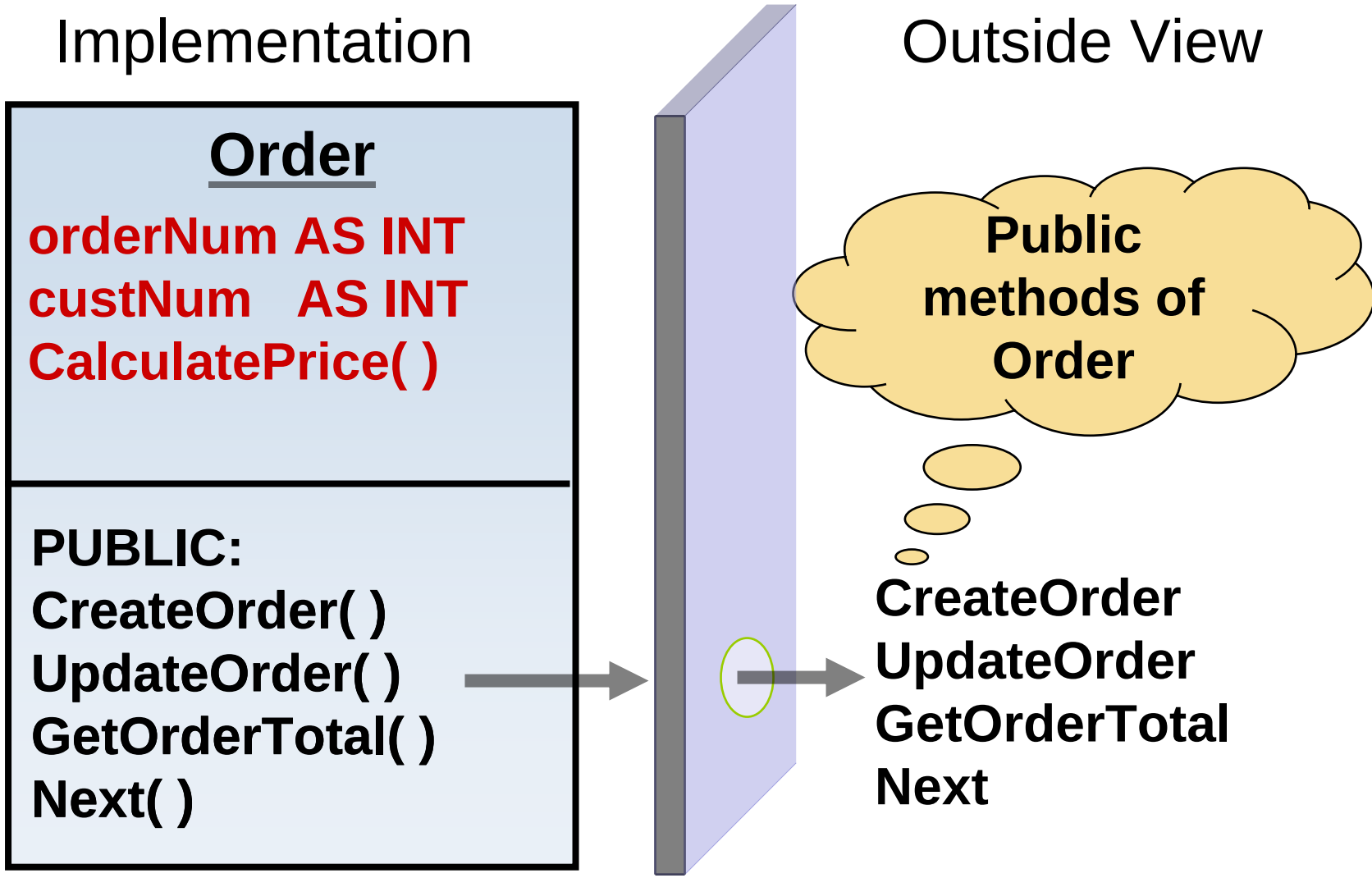



- Encapsulation is
  - The grouping of related ideas into a single unit, which can thereafter be referred to by a single name.
  - The process of compartmentalizing the elements of an abstraction that constitute its **structure** and **behavior**.
- *Encapsulation* hides implementation
  - Promotes modular software design – data and methods together
  - Data access always done through methods
  - Often called “information hiding”

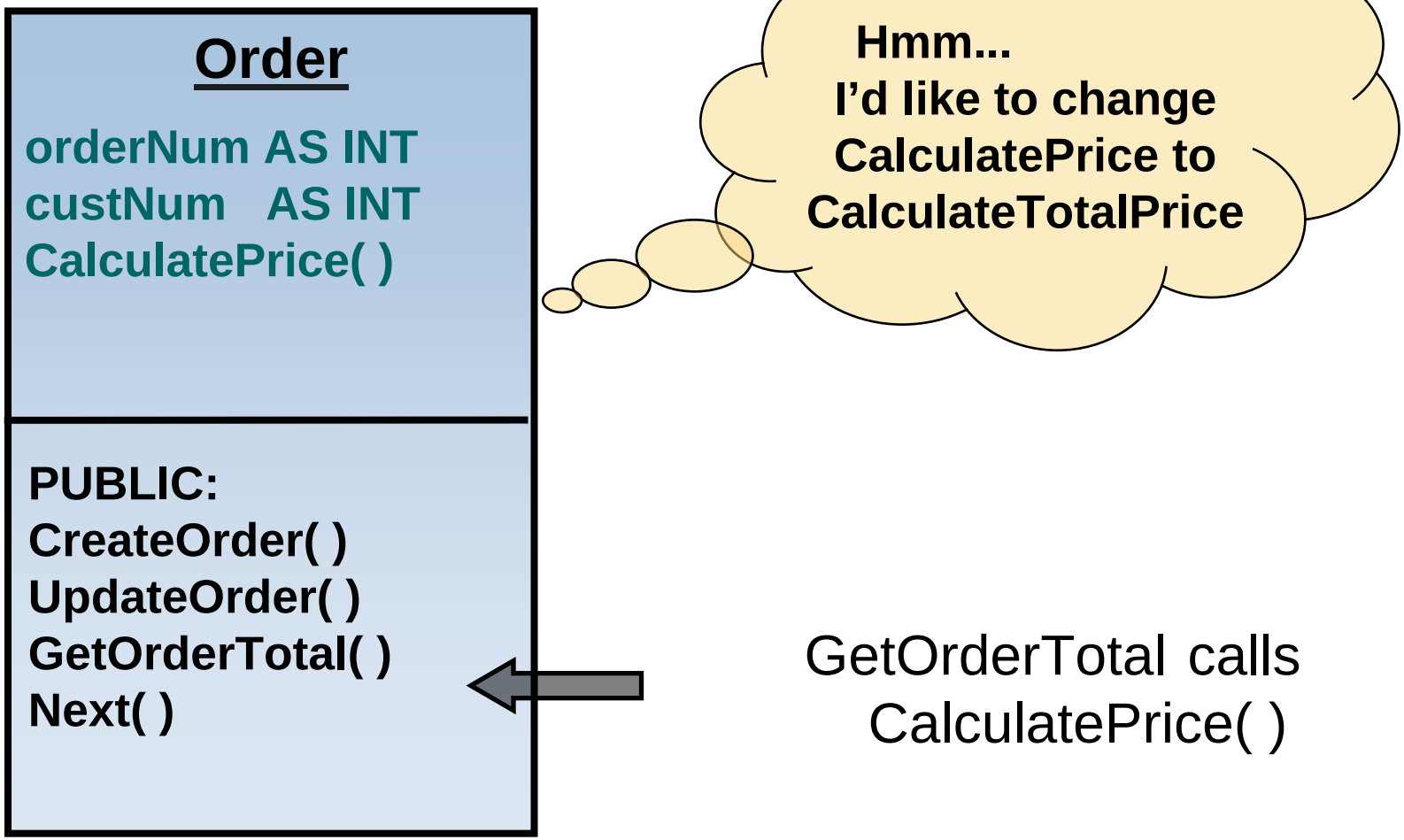


- Provides two kinds of protection:
  - State cannot be changed directly from outside
  - Implementation can change without affecting users of the object

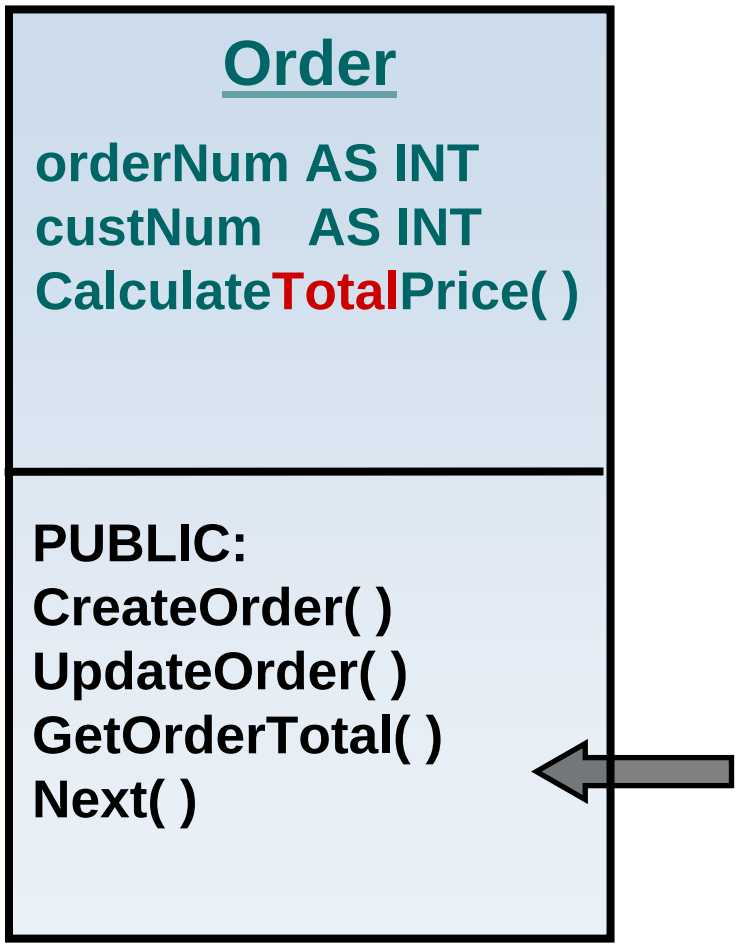
# Encapsulation - Example



# Encapsulation - Example continued



# Encapsulation - Example continued

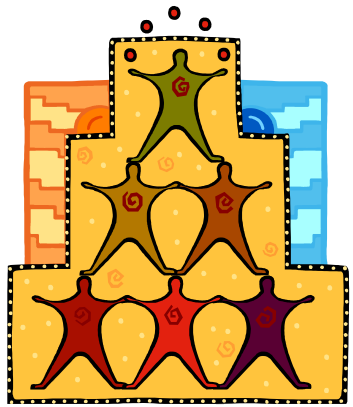


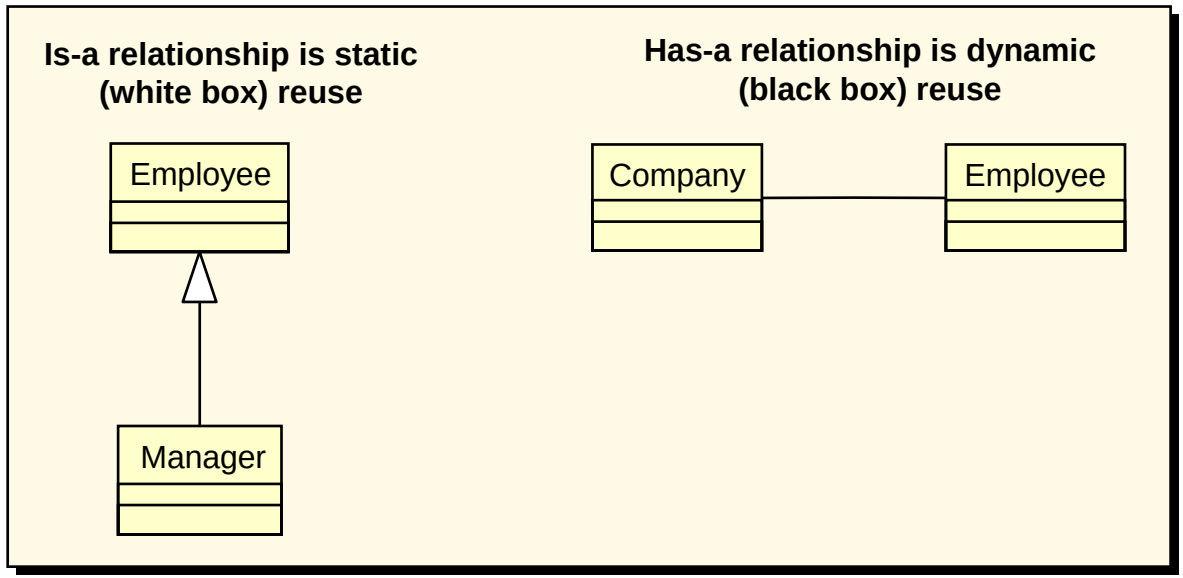

**This change was easy because users of the object will not be affected.**

GetOrderTotal now calls CalculateTotalPrice( )

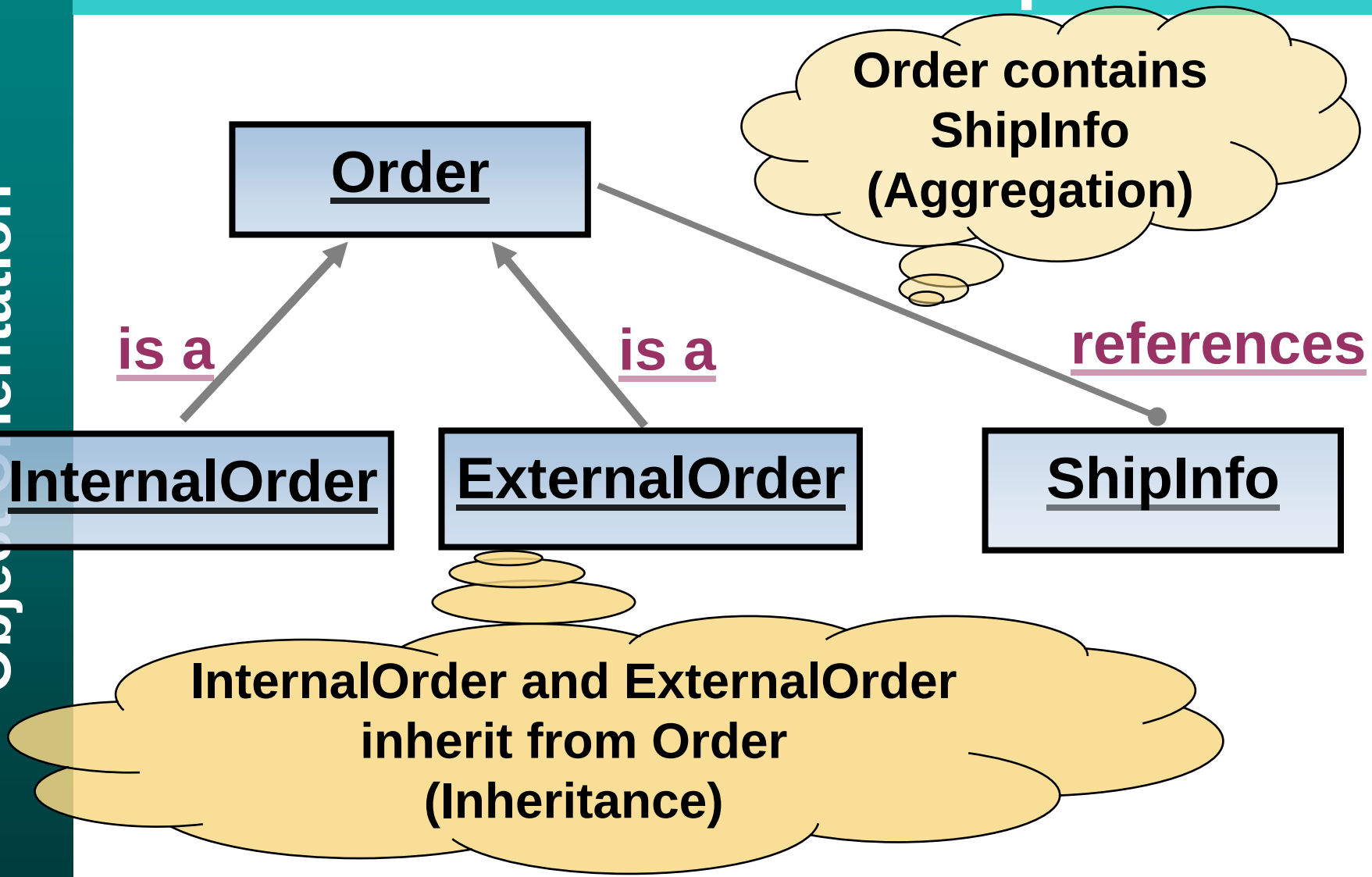
# Hierarchies- *Object Relationships*

- Define relationships between objects
- Objects defined in terms of other objects
- Allows state and behavior to be shared and specialized as necessary
- Encourages code reuse
- Two important hierarchy types:
- Inheritance (Is-a)
- Aggregation (Has-a)



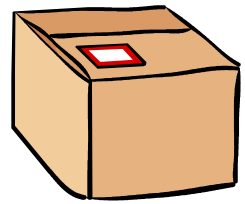


# Hierarchies - Example

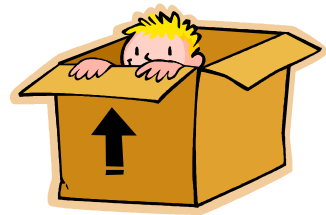


# Summary : Object-oriented Principles

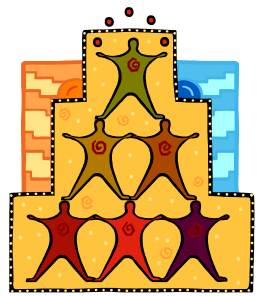
- Abstraction
  - Break up complex problem
  - Focus on public view, commonalities



- Encapsulation
  - Hide implementation details
  - Package data and methods together



- Hierarchies
  - Build new objects by referencing extending other objects



# Identifying Abstraction & Encapsulation

**Exercises**

Describe two of the following objects in terms of their attributes and operations:

motor car, sheep, kite, hospital, elephant, garden, school, bacon, teddy bear, bank customer, bus.

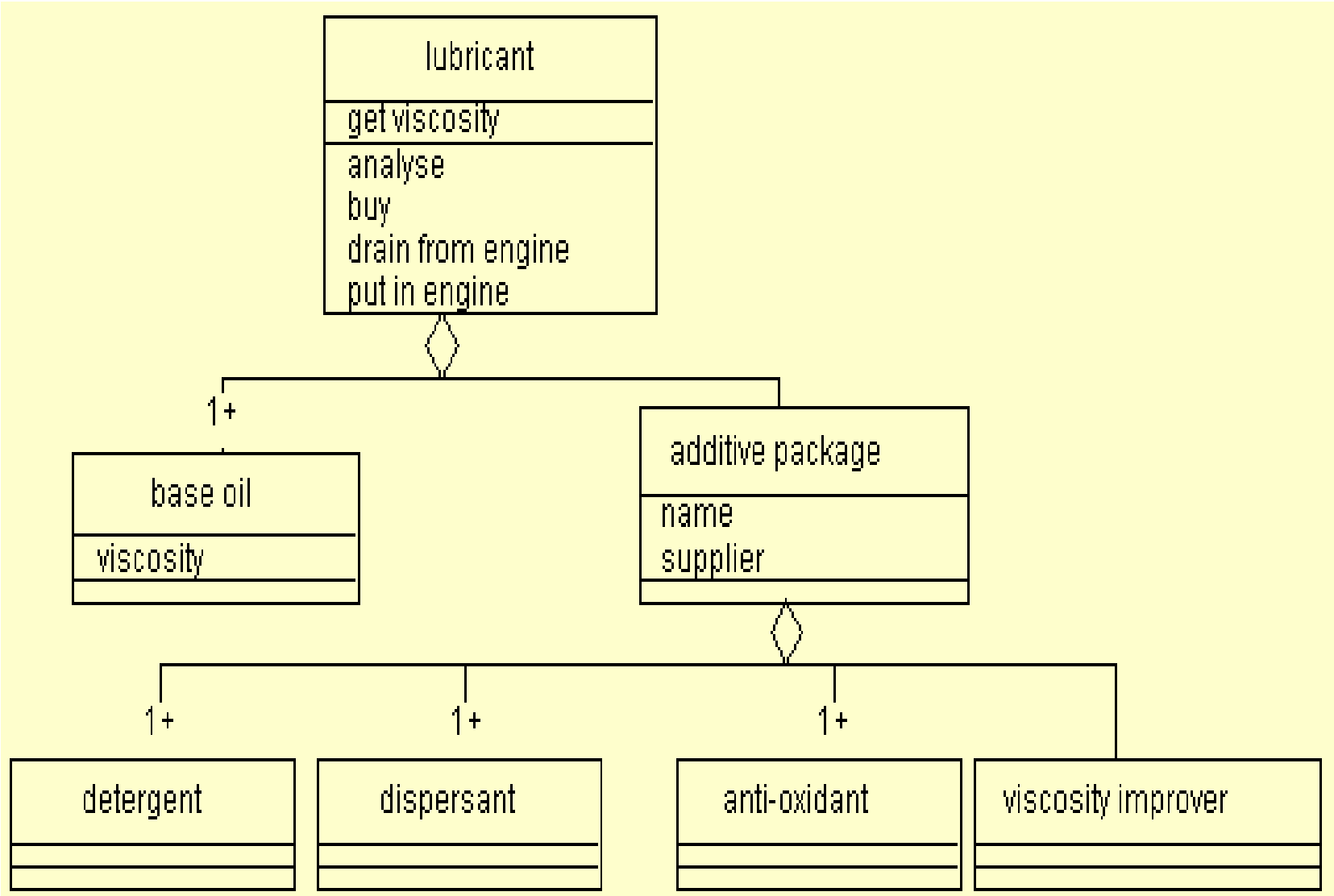
Devise a simple inheritance hierarchy for two of the following:

geometry, sports teams, politicians, rodents, viral infections, restaurants.

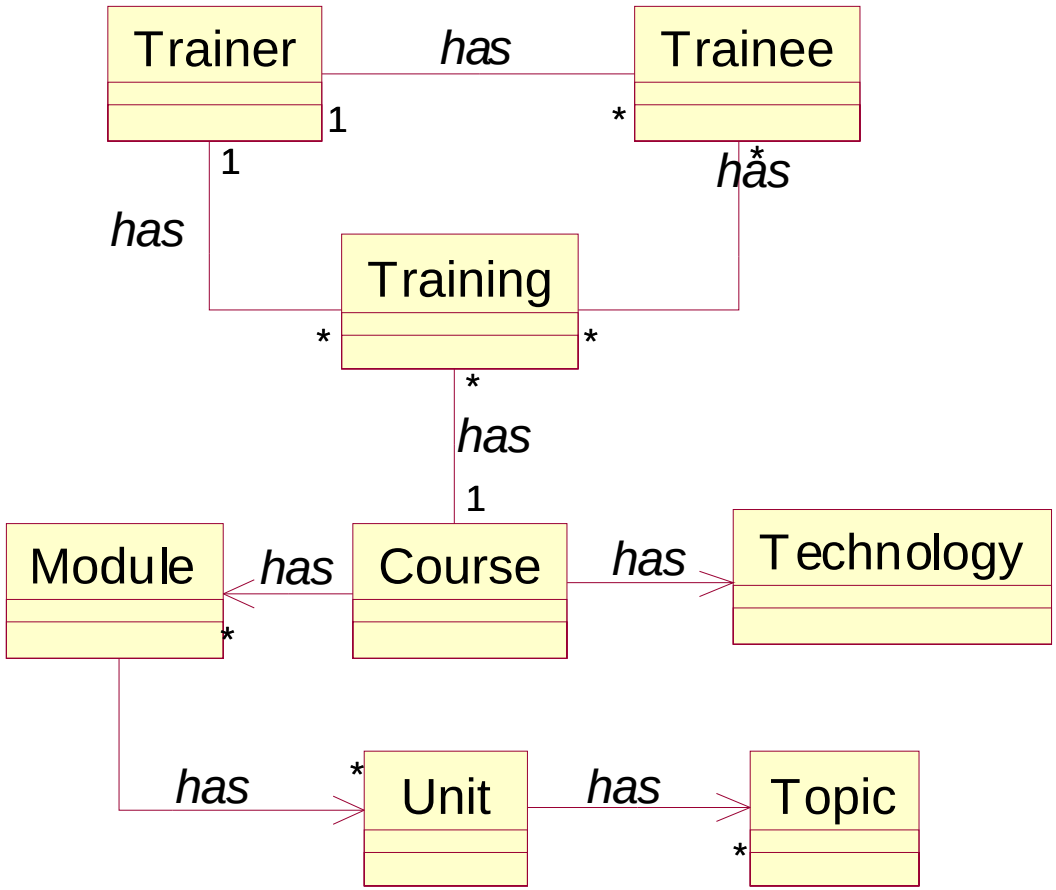
## Identify Abstractions, Encapsulations and Relationships

- A car engine lubricant is made up of a number of base oils blended with an additive package. The additive package consists of one or more detergents to keep engine surfaces clean, one or more dispersants to suspend particles in the oil to be carried to the filter, one or more anti-oxidants to slow up the thermal decay of the oil, and a viscosity improver to control the viscosity of the oil at different temperatures.
- A research scientist will experiment with different oils by running engines containing the oils and analyzing the effect of the engines on the oils.

## Identify Abstractions, Encapsulations and Relationships



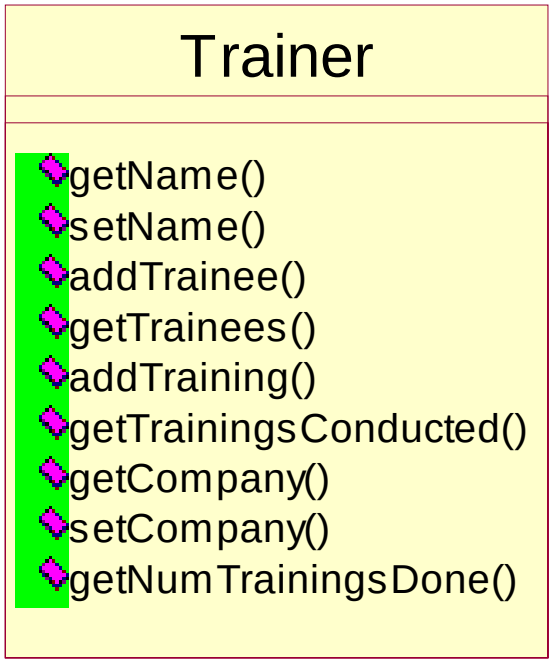
# Identifying Abstraction & Encapsulation




- Identify the abstractions (responsibilities) of all the classes
- Identify the Encapsulations (attributes & methods) for all the classes

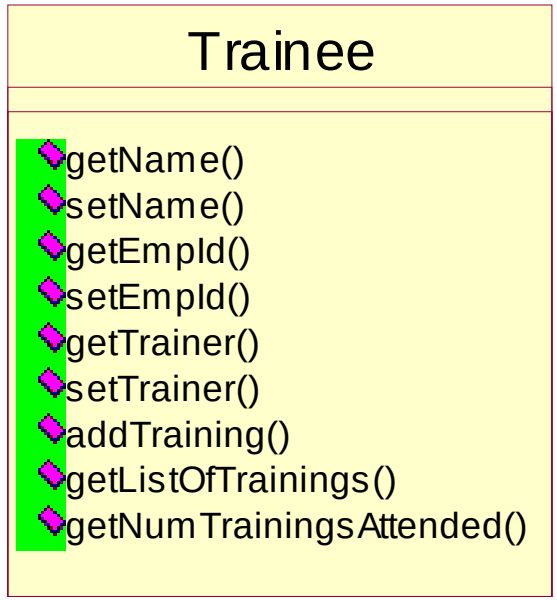
# Identifying Abstractions

- Abstractions for the Trainer Class



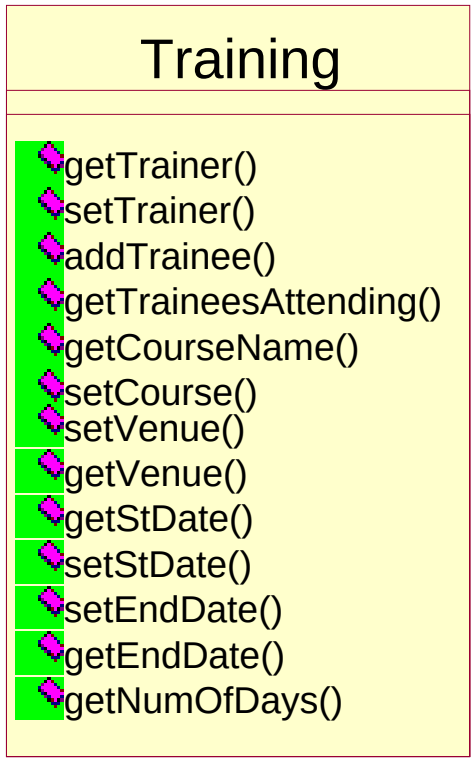
# Identifying Abstractions

- Abstractions for the Trainee Class



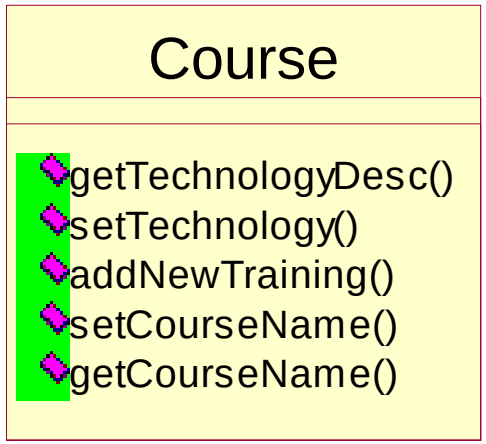
# Identifying Abstractions

- Abstractions for the Training Class

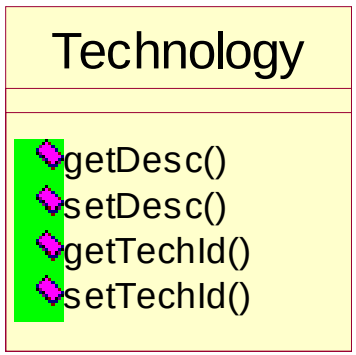


# Identifying Abstractions

- Abstractions for the Course Class

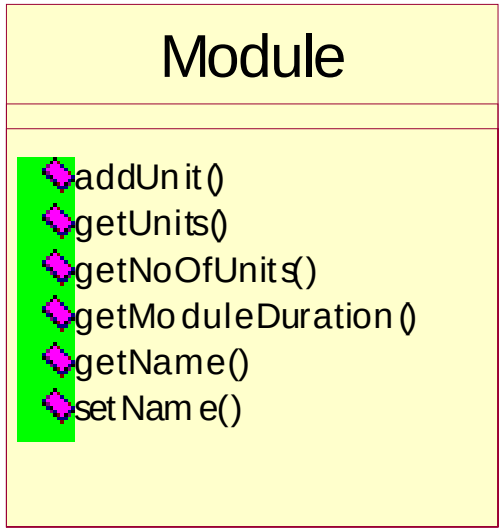



- Abstractions for the Technology Class



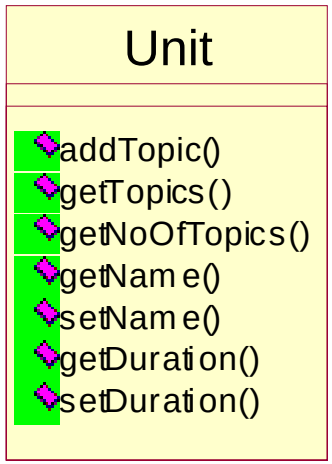
# Identifying Abstractions

- Abstractions for the Module Class

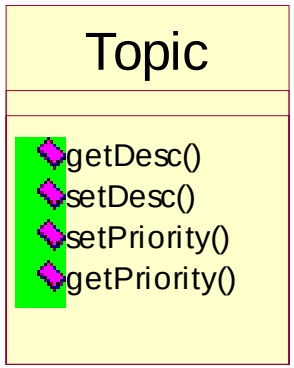


# Identifying Abstractions

- Abstractions for the Unit Class

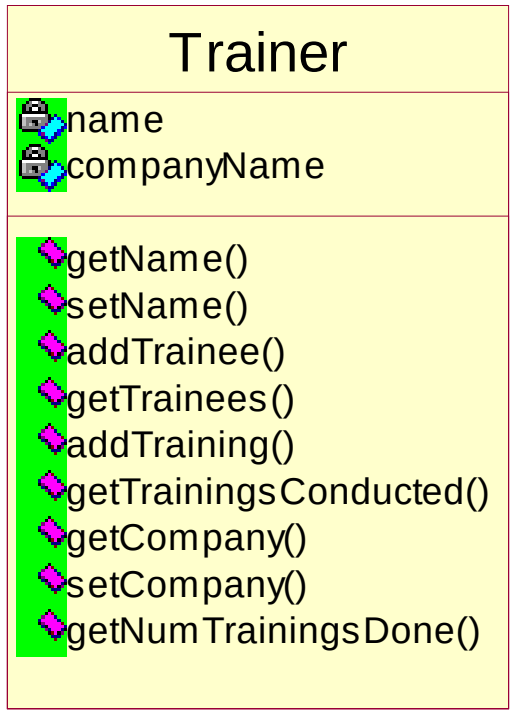



- Abstractions for the Topic Class



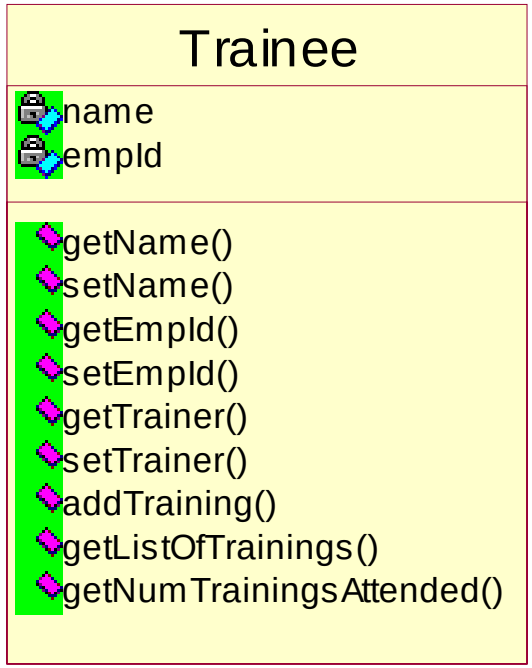
# Identifying Encapsulations

- Encapsulations for the Trainer Class



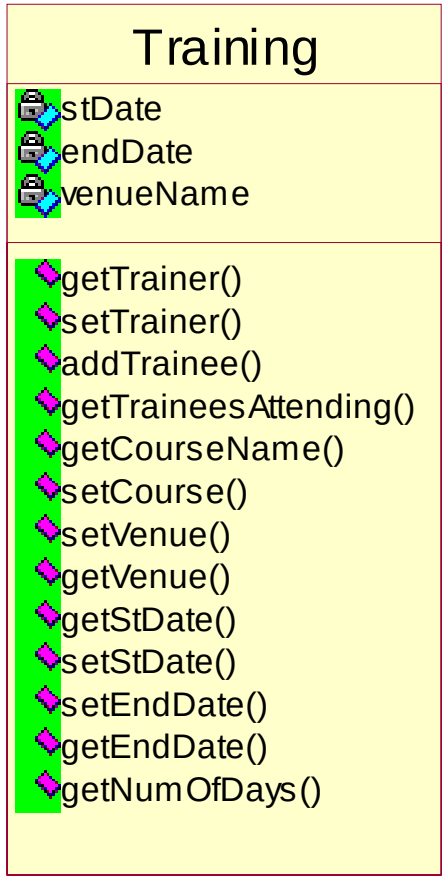
# Identifying Encapsulations

- Encapsulations for the Trainee Class



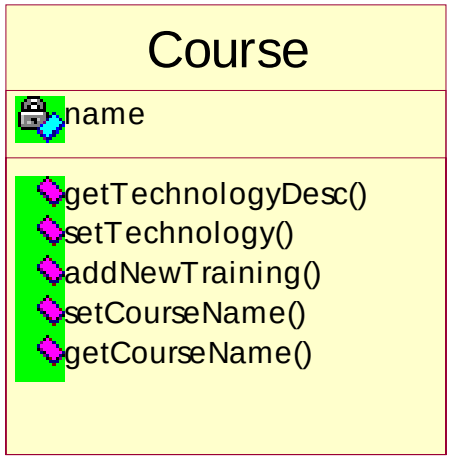
# Identifying Encapsulations

- Encapsulations for the Training Class

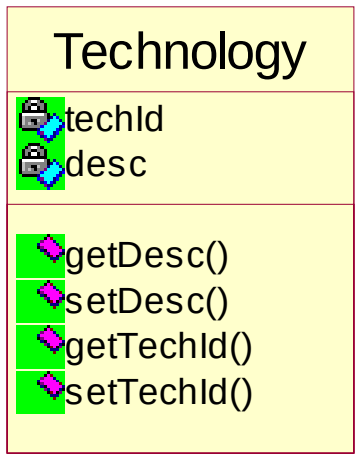


# Identifying Encapsulations

- Encapsulations for the Course Class



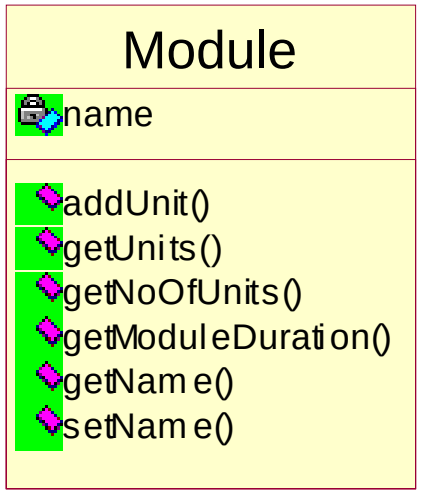

- Encapsulations for the Technology Class



# Identifying Encapsulations

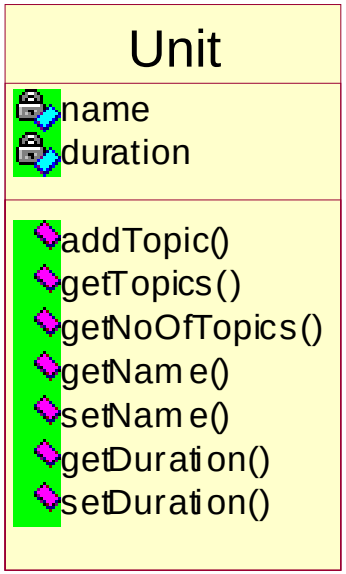


- Encapsulations for the Module Class

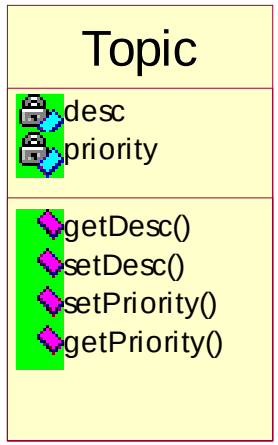


# Identifying Encapsulations

- Encapsulations for the Unit Class

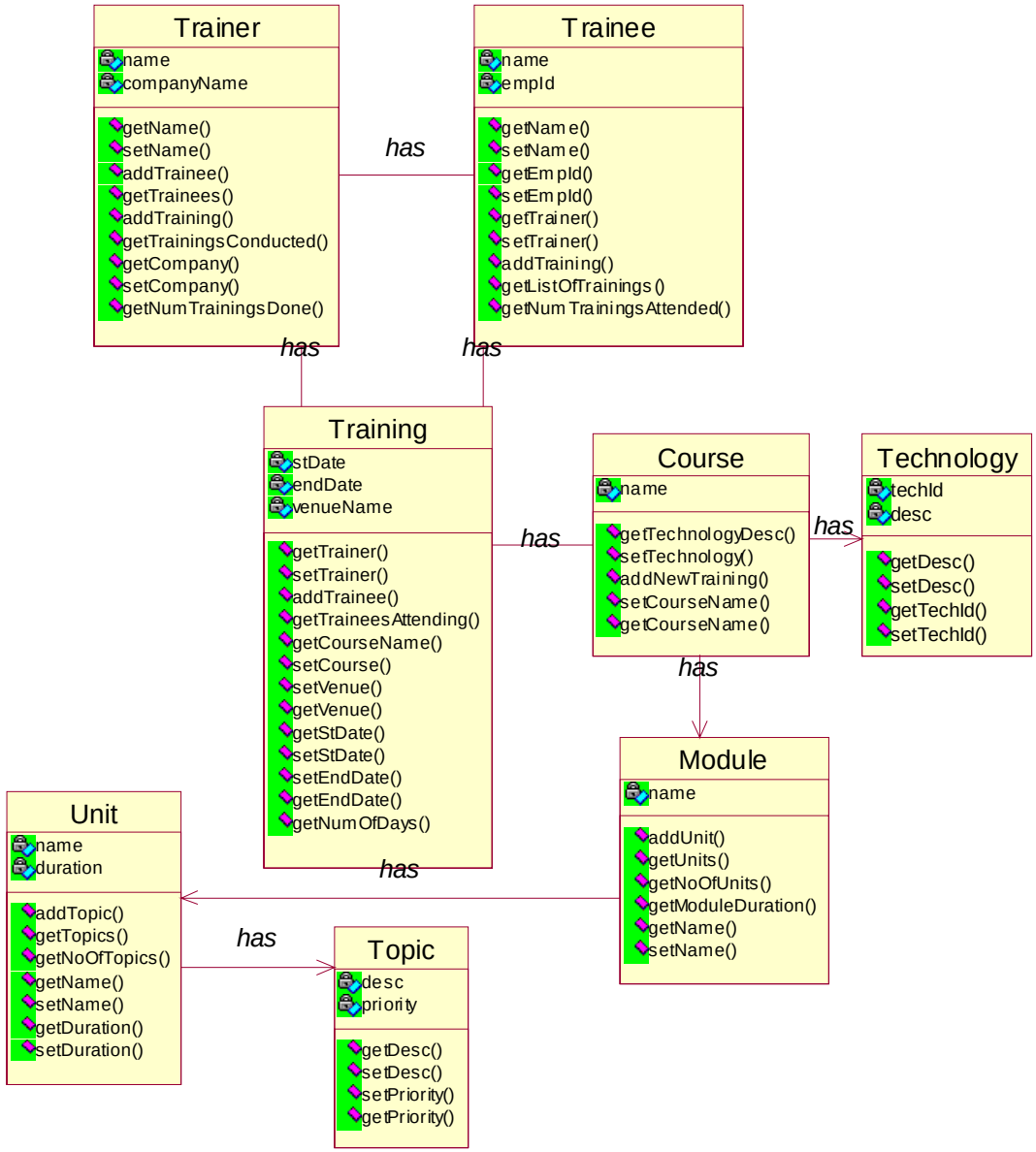



- Encapsulations for the Topic Class



Object Orientation

# Solution

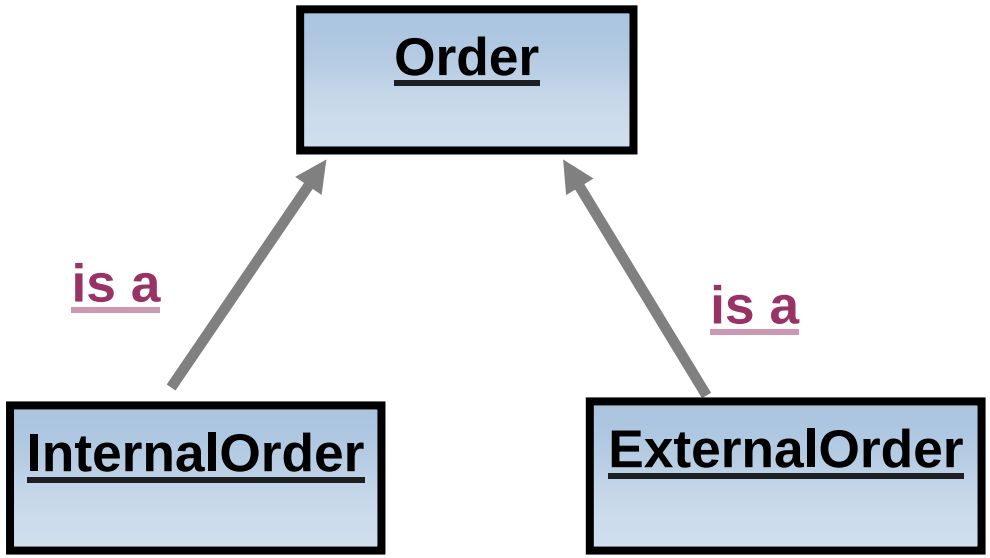


# Type

***A Type is a definition***

- A *Type* defines:
  - Data and methods
  - Inheritance relationships with other types
- No concern for implementation
- Enables strong-typing
  - Early (static) binding - types known at compile time
  - Type-consistency enforced at compile time and runtime

# Type - Example




Types:

- Order
- InternalOrder
  - Subtype of Order
- ExternalOrder
  - Subtype of Order

**A *subtype* can appear anywhere a *super type* is expected**

# Benefits of Types (Strong-Typing)

- Compile time checking for type consistency

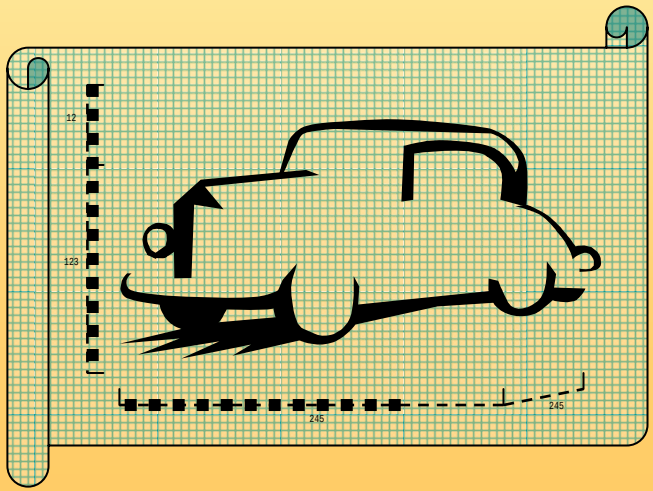
  myObj = mySubObject. (must be subType)
  myObj:method(…). (validates signature)
  myObj:data = 3. (validates data type)

- Results in safer, bug-free code because all code paths checked at compile time

# Comparing Classes to Objects

## Class

- A class is a template or blueprint that defines an object's attributes and operations and that is created at design time



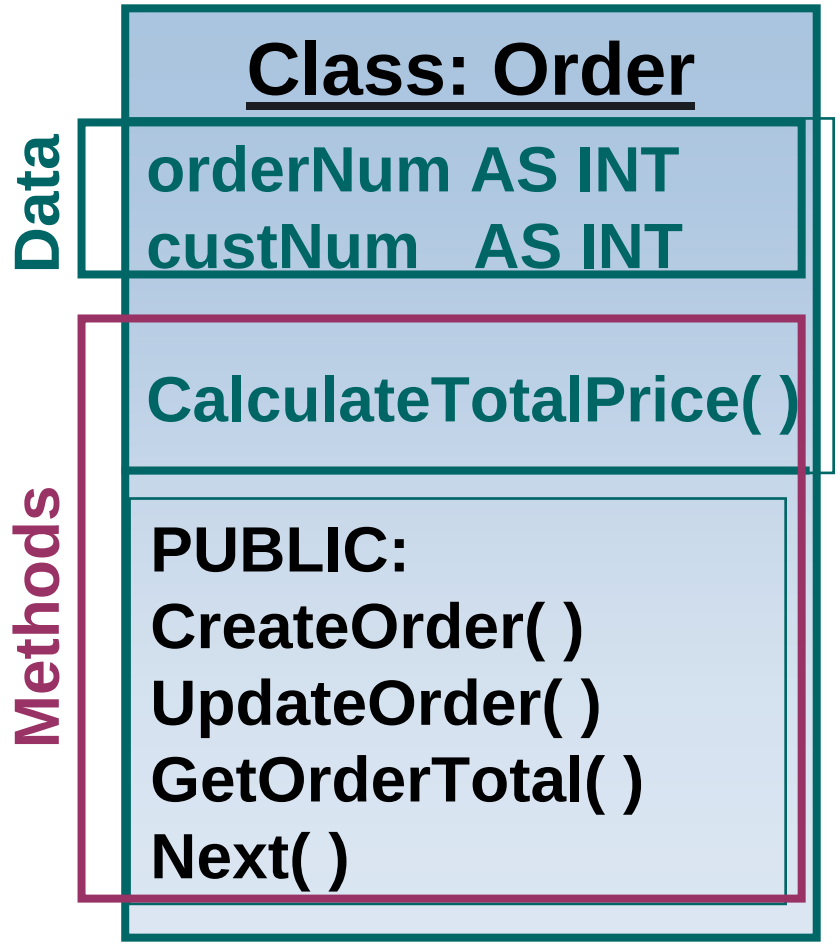
## Object

- An object is a running instance of a class that consumes memory and has a finite lifespan

# Class

***A Class implements a Type***

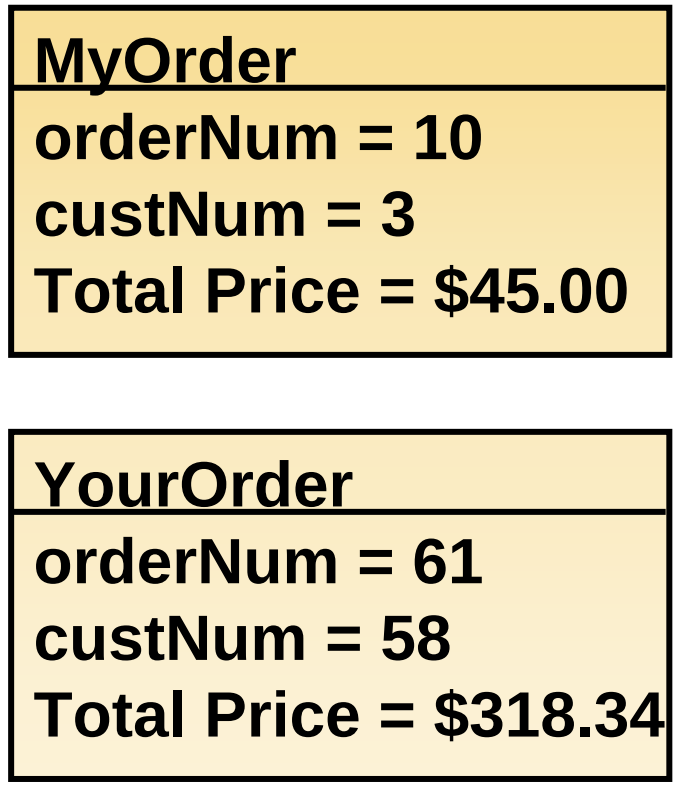
- A *Class* defines and implements a user-defined type
- A *Class* is a template (blueprint) for an object:
  - Data
  - Methods
  - Relationships to other classes



# Object

***An Object is an instance of a Class***

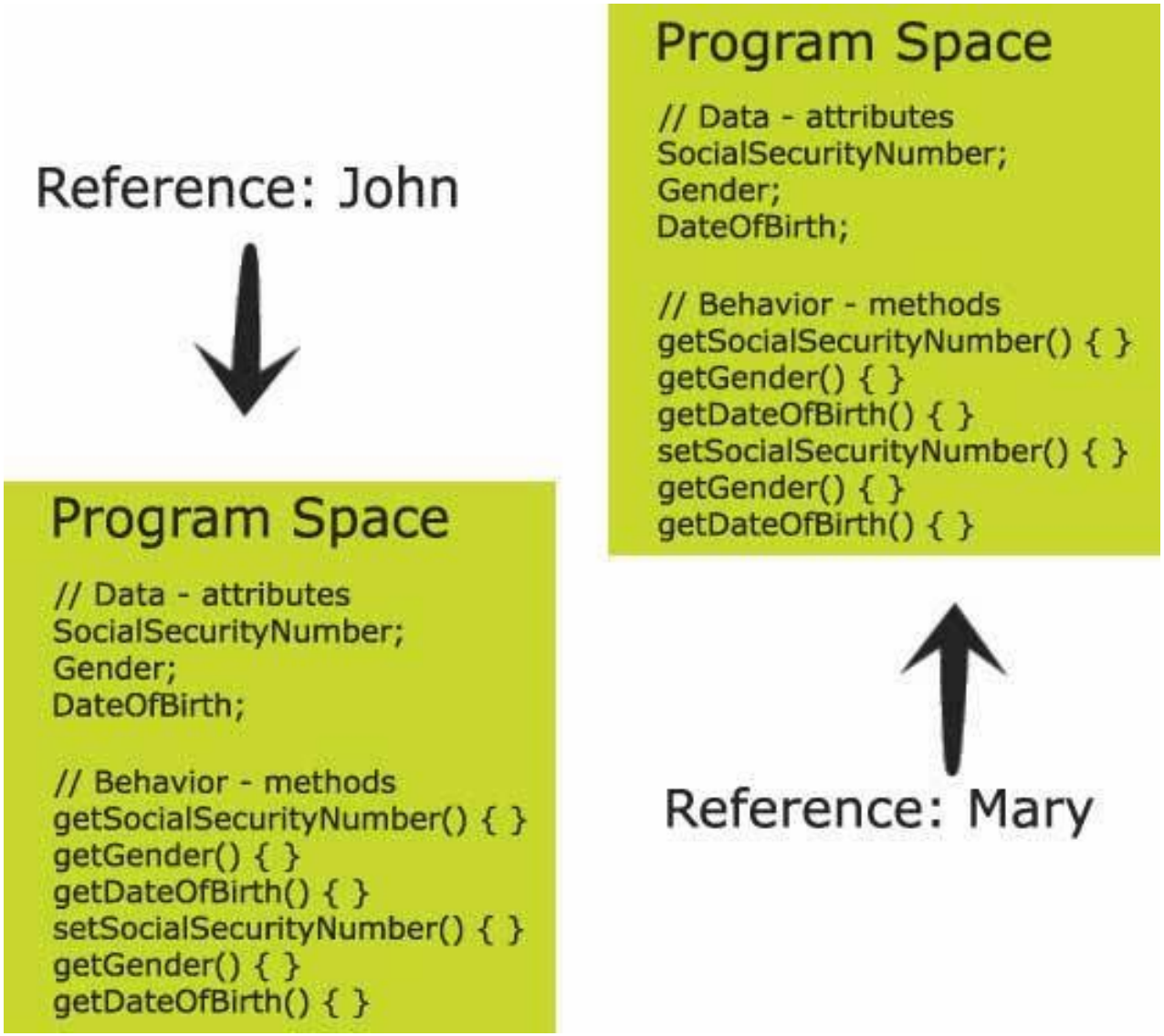
- An *Object* is created at runtime
  - Maintains independent state in data members
  - Code shared among object instances
- The term *Object* is often used to refer to both classes and instances



# Instantiation

- Instantiation is perhaps the most basic OO reusability mechanism.
- There are static and dynamic instantiation.
- Statically instantiated objects are allocated at compile time and exist for the duration that the program executes.
- Dynamically instantiated objects require run-time support for allocation and for either explicit deallocation or some form of garbage collection.

# Instantiation

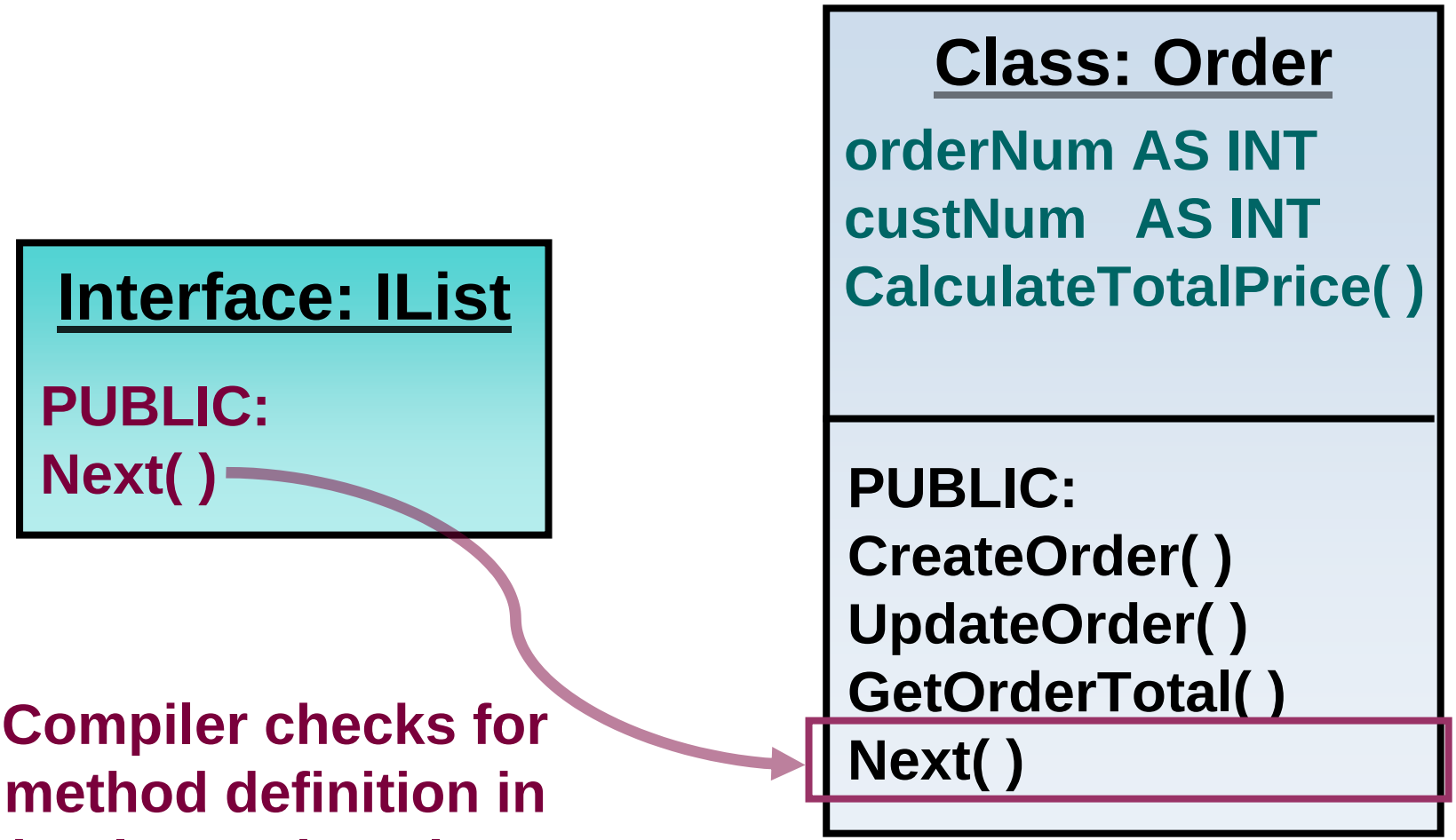


# Interface

***An Interface defines a Type***


- An *Interface* is a collection of method definitions for a set of behaviors – a "contract"
  - No implementation provided
- A *Class* can implement an interface
  - Must implement all methods in the interface
    - Behavior can be specialized
  - Compiler validates implementation of interface

# Interface - Example



# Interface – Example continued

- Write generic routine using interface

```
MoveNext( listObj AS IList )
    listObj.Next( ).                    /* Calls method
    …                                      in real object */
```

- Call with any object that implements IList

```
myOrder = NEW Order( ).          /* implements IList */
MoveNext( myOrder ).      /* Order's Next()   */
or
myCust = NEW Customer( ). /* implements IList */
MoveNext( myCust ).       /* Customer's Next()*/
```

# Benefits of Interfaces

- Allows many different classes to be treated in a like manner
  - All classes that implement an interface are guaranteed to have same set of methods
  - Enables generic programming
    - IList example allows any collection of objects to be navigated using Next( )
  - Behavior can be specialized as needed

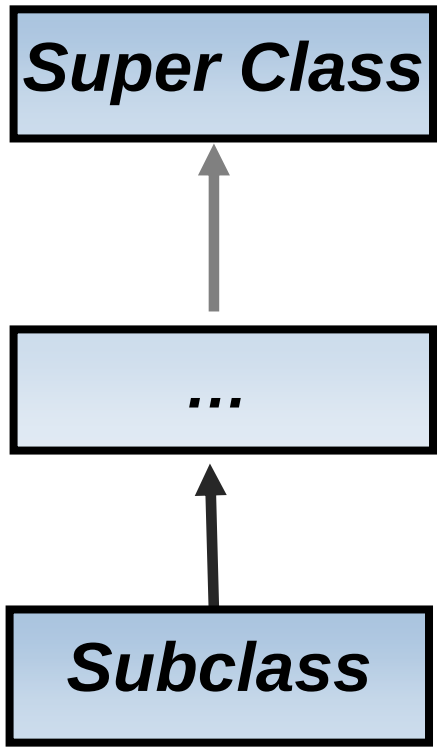
# Summary

- Type
  - Enforces type consistency at compile time
- Class
  - Defines type with data and methods and provides implementation
- Object
  - Runtime instantiation of class
- Interface
  - Defines type with only methods – no implementation provided

# Inheritance

***Relationship between Classes***

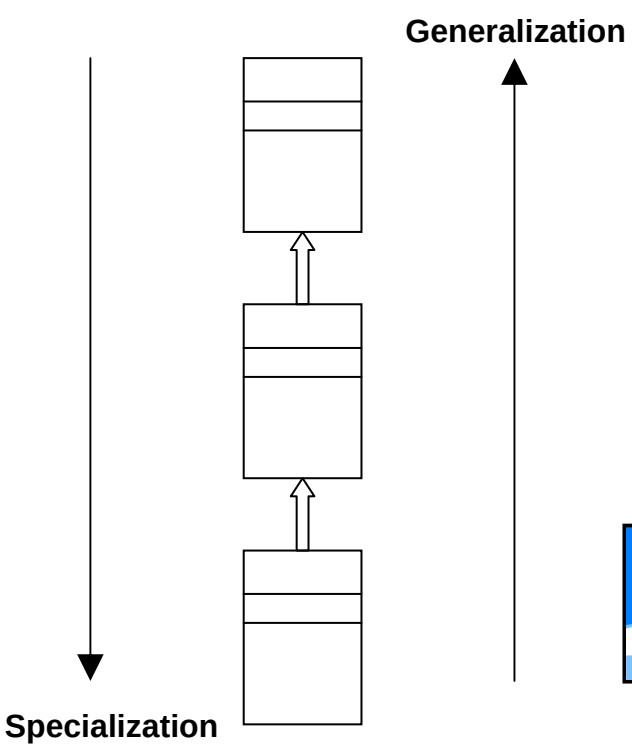
- *Super Class* (Base class)
  - Provides common functionality and data members
- *Subclass* (Derived class)
  - Inherits public and protected members from the *super class*
  - Can extend or change behavior of super class by *overriding* methods

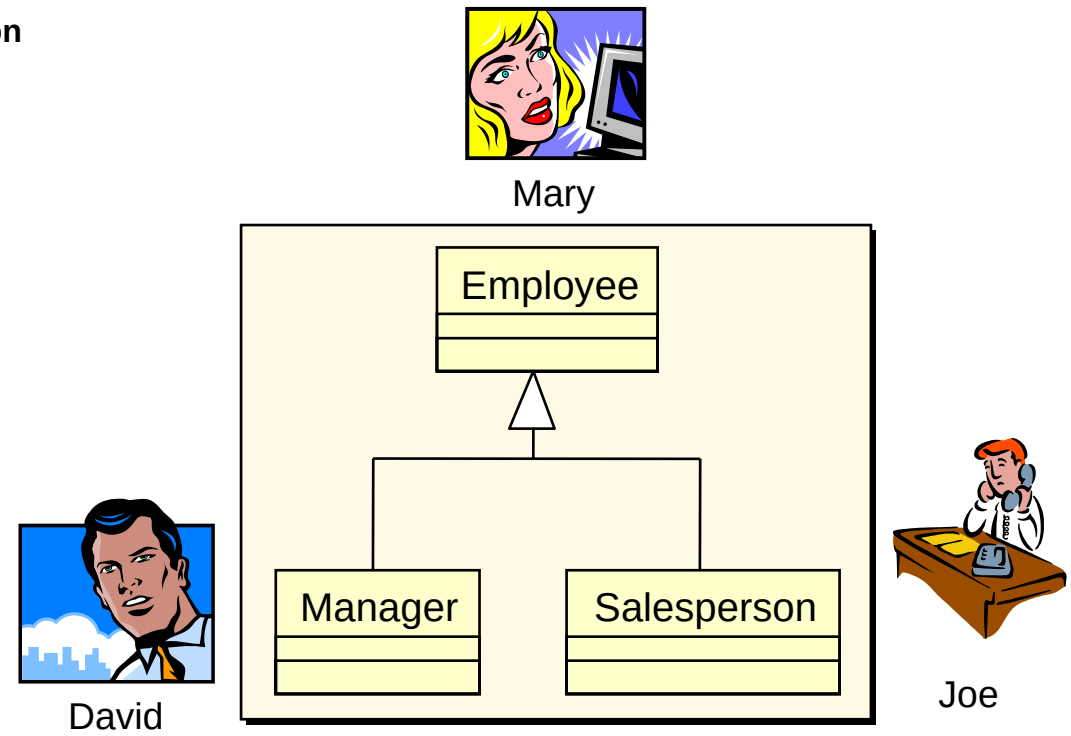


# Inheritance

- An object oriented system organizes classes into a subclass-super class hierarchy

Each subclass has added responsibilities
Can implement behavior in different ways

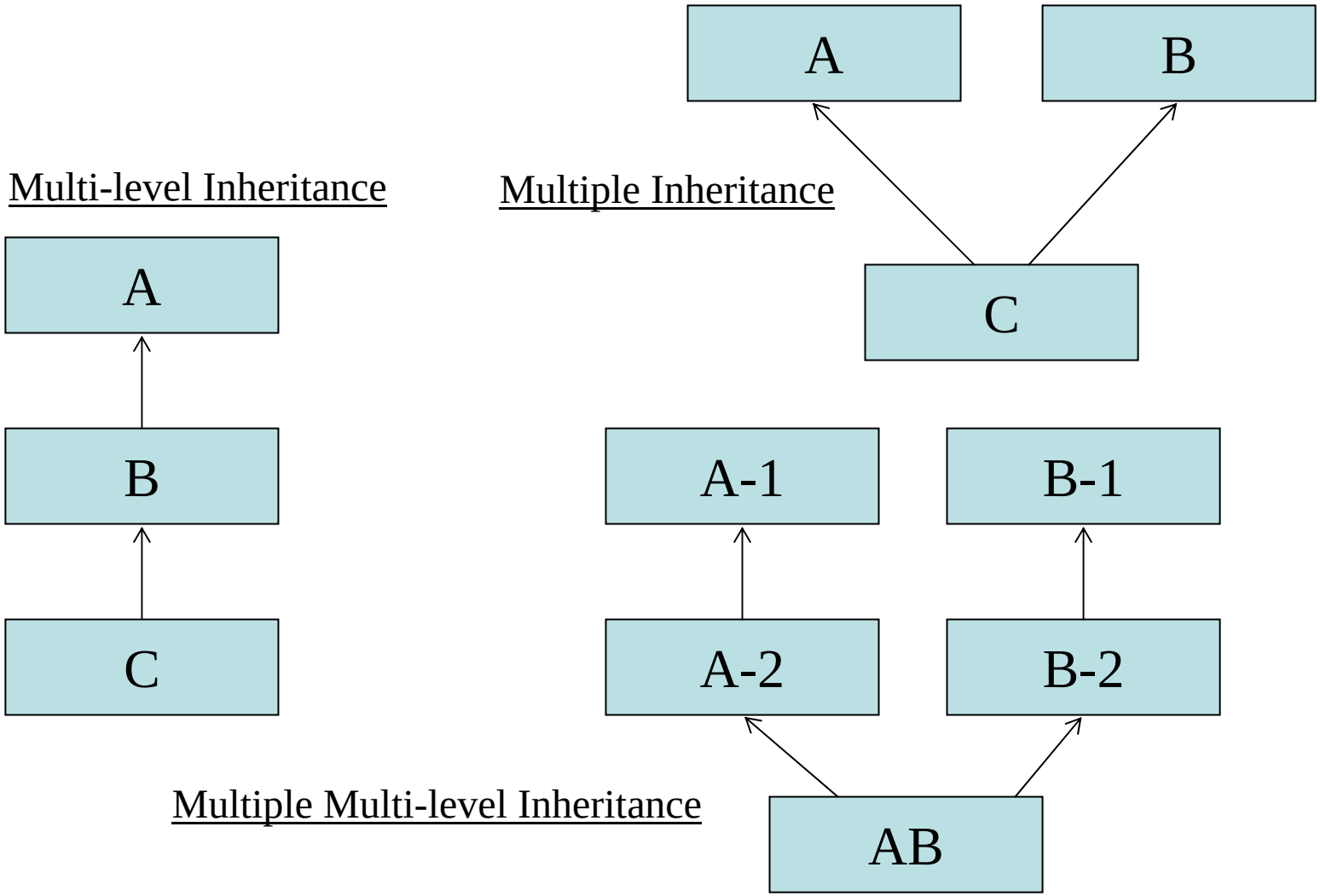


# Inheritance

- In most cases, inheritance is strictly a reusability mechanism for sharing behavior between objects.
- Class inheritance is often represented as the fundamental feature that distinguishes object-oriented from other programming languages.
  - Single inheritance
  - Multiple inheritance.

# *Types of Inheritance:*



## Some interesting points concering class inheritance

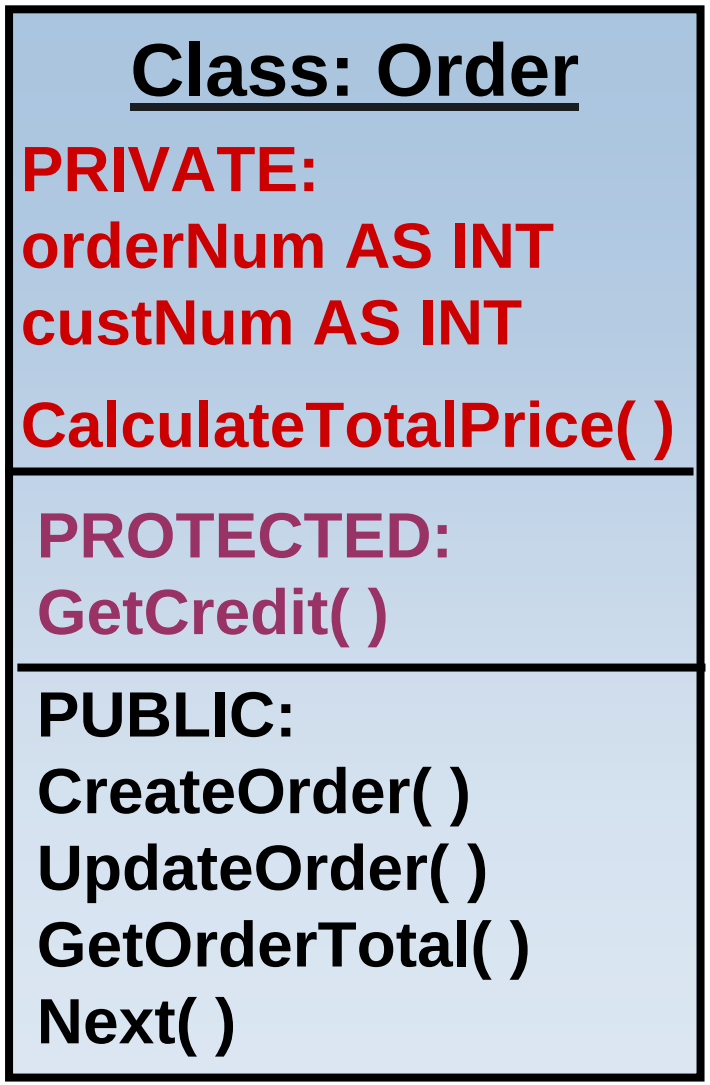
- Not all languages with class inheritance support multiple inheritance.
- It is important to be able to override inherited methods
- Subclass may or may not be permitted direct access to inherited instance variables.
- The issue of name clashes in the presence of multiple inheritance.

# Access Levels for Class Members

- *PRIVATE* members available:
  - Only within the class
- *PROTECTED* members available:
  - Within the class
  - Within the class hierarchy
- *PUBLIC* members available:
  - Within the class
  - Within the class hierarchy
  - To users outside the class

| Order |
| --- |
| PRIVATE |
| PROTECTED |
| PUBLIC |

# Inheritance Example



# Inheritance Example

**Class: Order**

**PRIVATE:**
**orderNum AS INT**
**custNum AS INT**

**CalculateTotalPrice( )**

**PROTECTED:**
**GetCredit( )**

**PUBLIC:**
**CreateOrder( )**
**UpdateOrder( )**
**GetOrderTotal( )**
**Next( )**

Class InternalOrder inherits Order

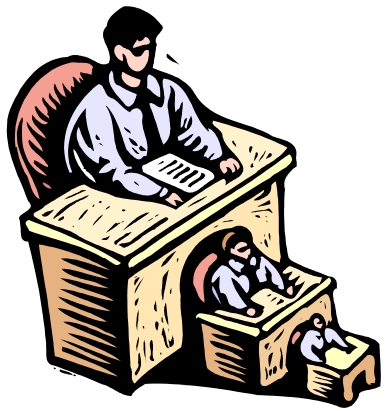
**InternalOrder**

**PROTECTED:**
**GetCredit( )**

**PUBLIC:**
**CreateOrder( )**
**UpdateOrder( )**
**GetOrderTotal( )**
**Next( )**

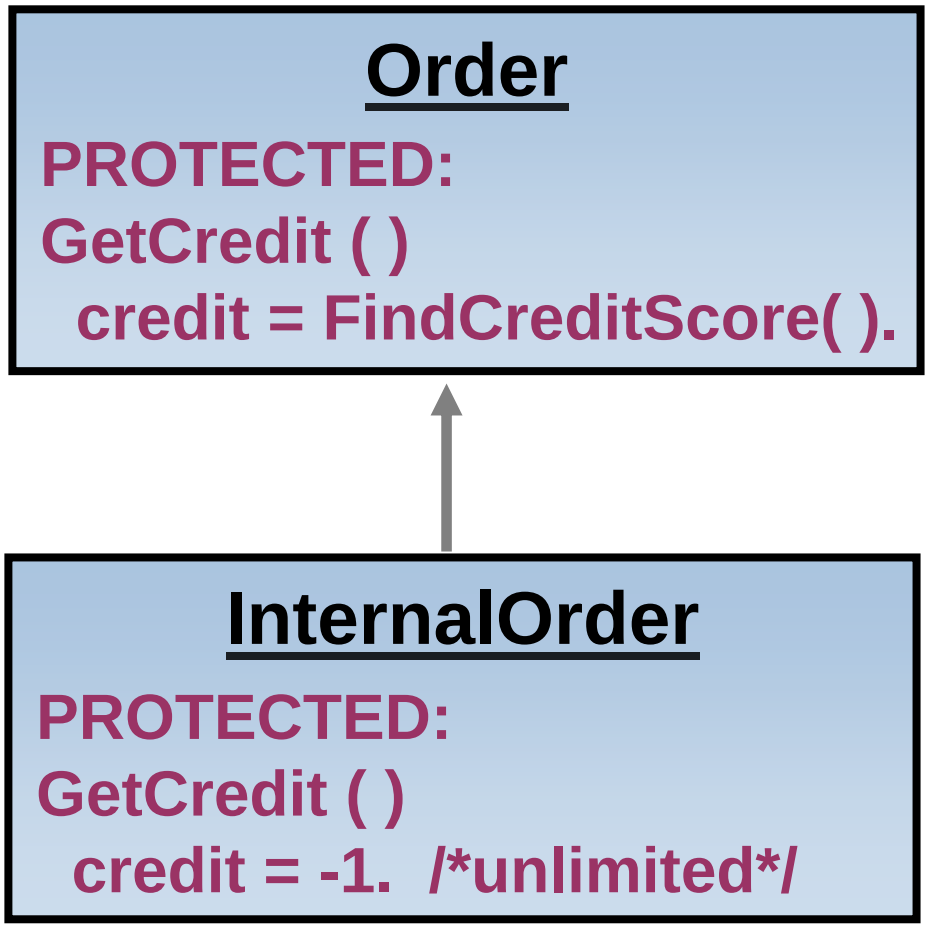
# Inheritance and Method Overriding

- Method *overriding* used to specialize behavior
  - Subclass may override a method in its super class (hierarchy)
  - Method signatures must match
- Overriden method can:
  - Completely override behavior of super class
  - Augment behavior by providing its own behavior and calling super class method



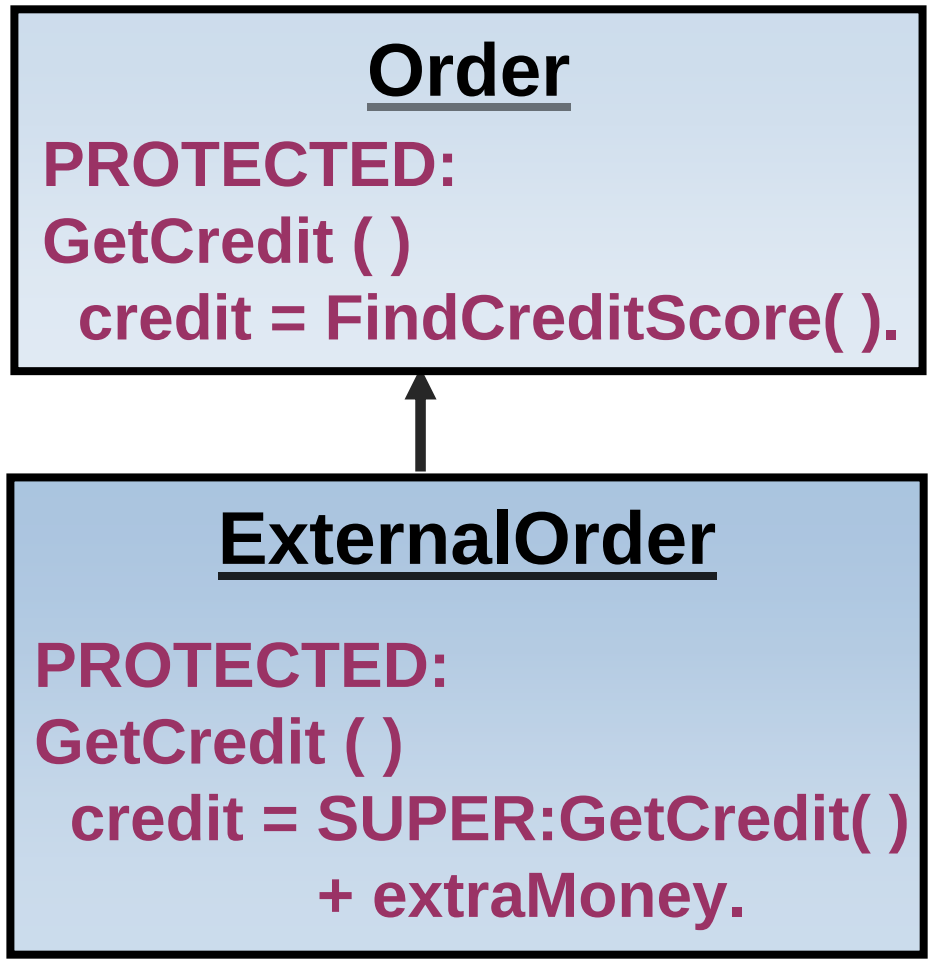
# Method Overriding – Example 1

Class
InternalOrder
inherits Order



# Method Overriding – Example 2

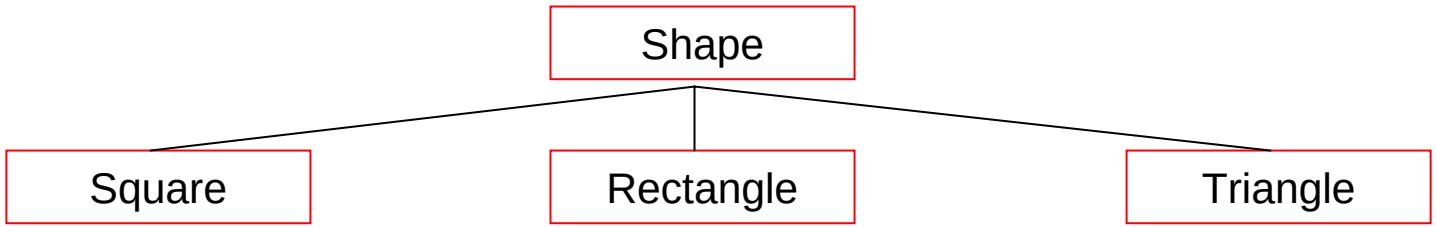
Class ExternalOrder inherits Order



# Method Overriding – Example 3

Problem:

A shape class contains the length of two dimensions and the behaviour *calculate_area* as attributes. Three sub-classes of this class are defined as Square, Rectangle and Triangle.



# Method Overriding – Example 3 (cont..)

- The *calculate_area* behaviour returns the area of the shape by multiplying one dimension by the other.

What problem arises with this class hierarchy ?

- The inherited behaviour *calculate_area* will provide an incorrect answer for an object of class Triangle.

# Method Overriding – Example 3 (cont..)

- To overcome this problem we can remove the common behaviour from the class shape and incorporate separate behaviours in all three sub-classes Square, Rectangle and Triangle.

- Each sub-class class can contain the same behaviour named *calculate_area*. This is known as polymorphism.

- This method is perfectly legal but there is some duplication in behaviour particularly between the Rectangle and Square classes.

# Method Overriding – Example 3 (cont..)

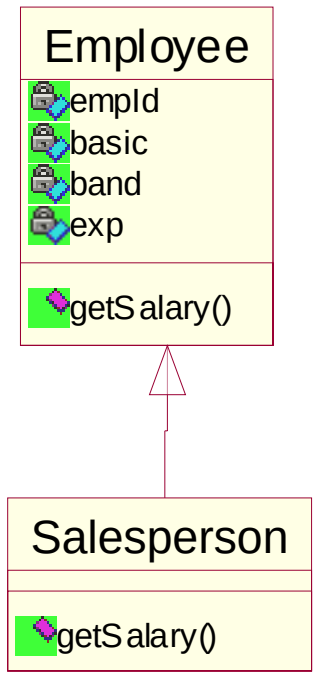
- Fortunately Object-Orientated Design provides a more efficient and natural way to overcome this problem.

- A subclass can override the behaviour defined in a super-class i.e. if a specific behaviour is defined in a class as well as its super-class then the class behaviour takes priority.

- e.g. the generic behaviour *calculate_area* can be inherited by the Square and Rectangle classes but we can override the *calculate_area* behaviour in the Triangle class.

# Benefits of Inheritance & Overriding

- Inheritance supports modular design
- Common behavior put in super class and used by subclass
- Subclass can override to specialize behavior



# Polymorphism

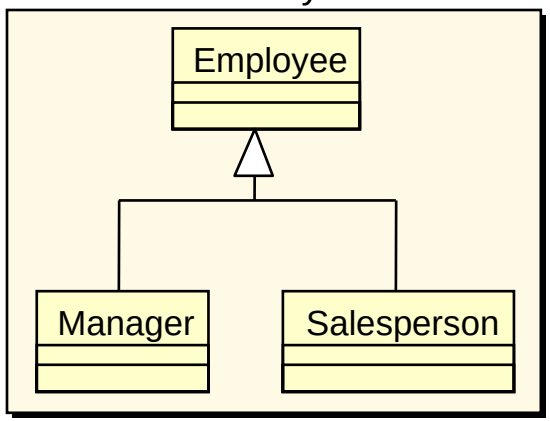
***One interface, many implementations***

- *Polymorphism* is the ability to call an overridden method in a subclass using a super class object reference
  - Code compiled using super class reference
  - Runtime dispatches method call to subclass
- Three requirements:
  - Hierarchy with overridden method in subclass
  - Super class reference used to call method
  - Subclass assigned to super class reference

# Polymorphism

All employees do some work

Mary




David

Joe

David does a manager's work

Joe does a salesperson's work

**Early binding**

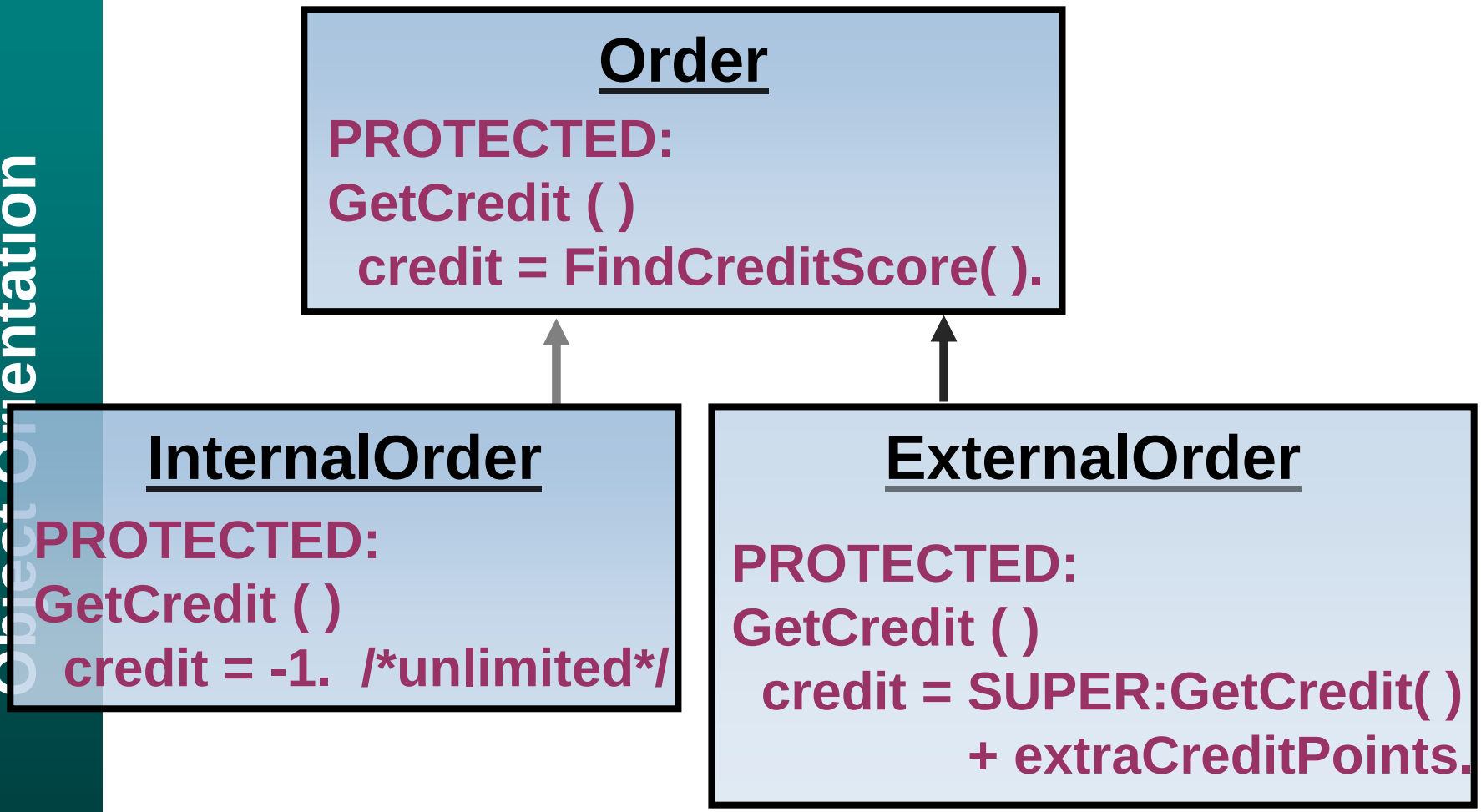
- Function Call mapped at compilation
- Function Overloading
- Class type and object type are the same

**Late binding**

- Function Call mapped at run-time
- Function Overriding
- Class type and object type need not be the same

# Polymorphism – Example





# Polymorphism – Example continued

```
DEFINE myOrder AS Order.

if (bInternalCust = TRUE)
    myOrder = NEW InternalOrder( ).
else
    myOrder = NEW ExternalOrder( ).

myOrder:GetCredit( ).
```

**Super Class reference**

**Calls InternalOrder:GetCredit( ) or ExternalOrder:GetCredit( )**

# Benefits of Polymorphism

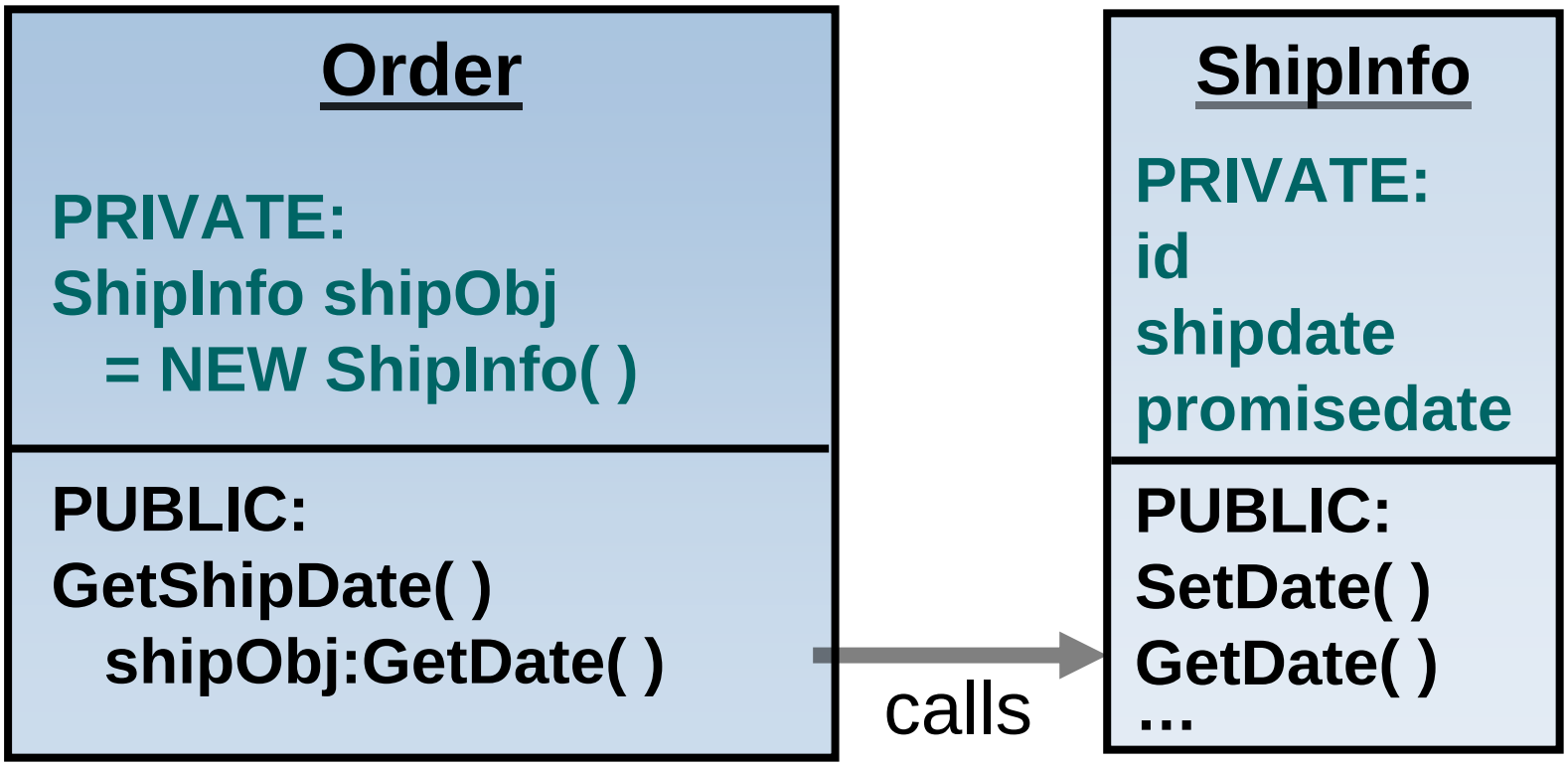
- Supports generic programming using super class or interface
  - Type used at compile time is super class or interface
- Specialized behavior is called at runtime automatically
  - Built on inheritance and overriding

# Delegation

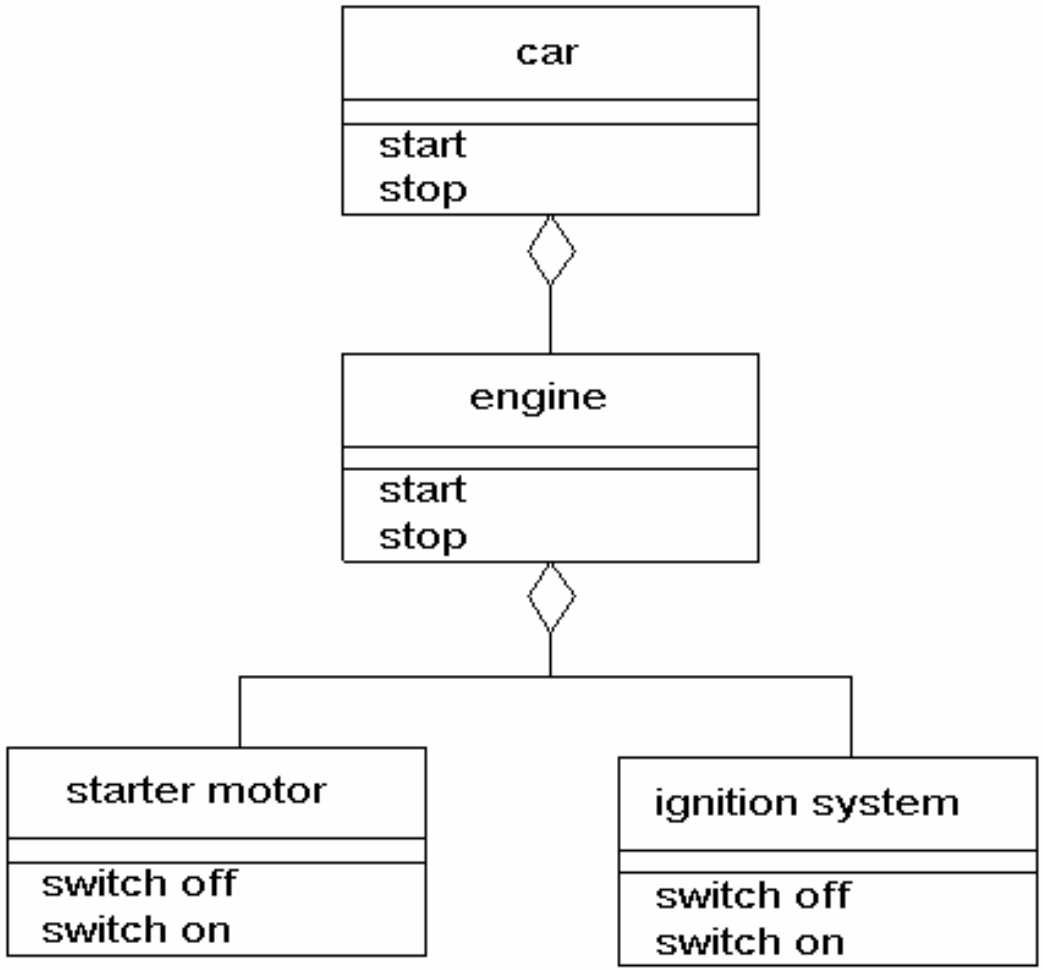
- *Delegation* is the use of other class within a class
  - Class forwards method calls to the contained class
- Class wraps the *delegate* class
  - Creates an instance of the class
  - Defines a “stub” method for any referenced methods that should be public
  - No access to protected or private members

# Delegation – Example

## Order references a ShipInfo object



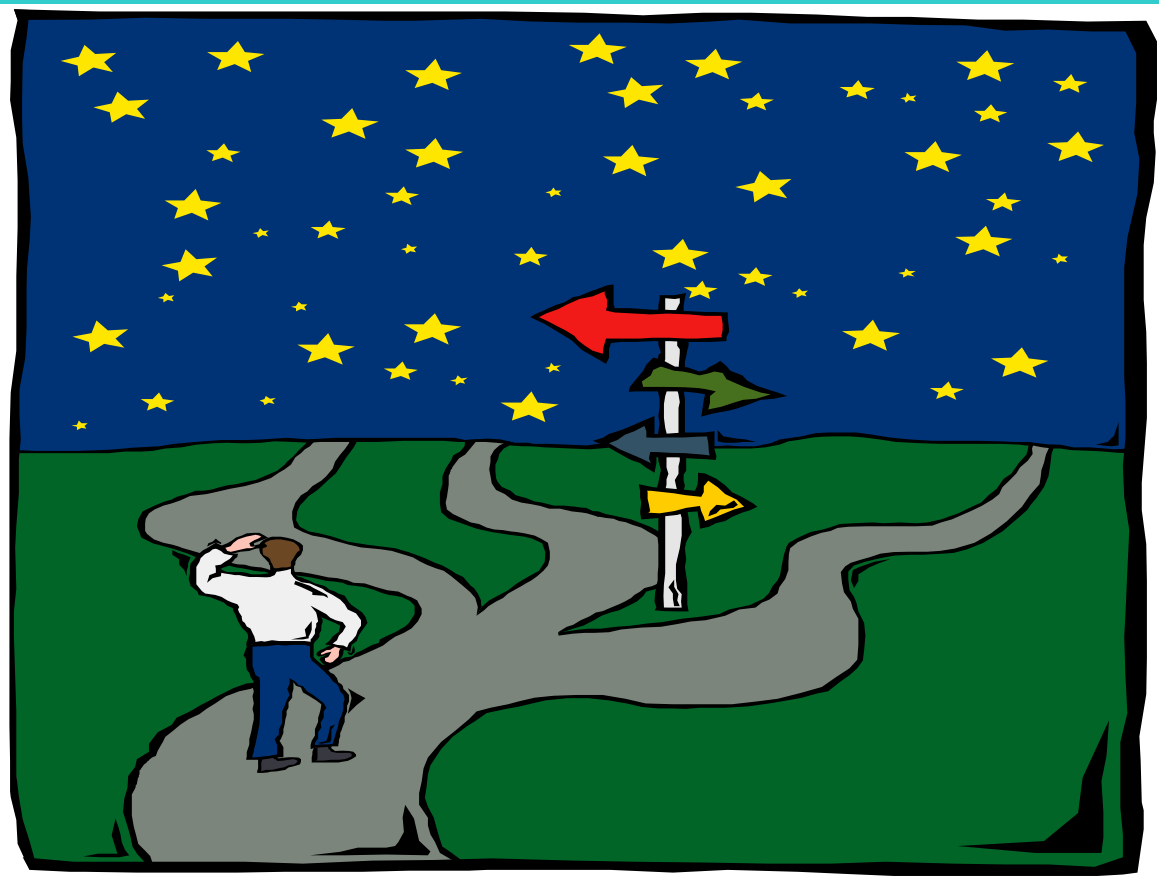
# Delegation – Example



# Benefits of Delegation

- Delegation supports modular design
  - Class uses delegate to provide needed functionality
- Class can determine what to put in API
  - With inheritance super class dictates API;
  - with delegation wrapper class decides what to expose

# That's Object-oriented Programming What did we learn…



# Terminology / Concept Review

- Abstraction – Public API
- Encapsulation – Hide implementation details
- Hierarchy – Relationships between classes

- Type – Strong-typing at compile time
- Class – Defines type; data and methods
- Object – Runtime instance of a class
- Interface – Set of method definitions; contract

- Inheritance – Inherit/specialize from super class
- Polymorphism – Most-derived method called from super class reference
- Delegation – Other objects do the work

# Benefits of OO Programming

- Promotes modular design
  - Data and methods that operate on that data are contained in one place
  - Commonalities put into super classes
- Code reuse through hierarchies
  - Inheritance and delegation
- Strong-typing
  - Compile-time type checking
  - Runtime type checking

# Question Time

Please try to limit the questions to the topics discussed during the session.

Participants can clarify other doubts during the breaks.

Thank you.