# Lecture 3

Abstraction
Aggregation and Composition
Polymorphism and Binding

# Aggregation and Composition

## Introduction to the Static Model

"Complexity *is* the business we are in, and complexity is what limits us."—Frederick Brooks

# Abstraction and Inheritance

# Abstract Data Types

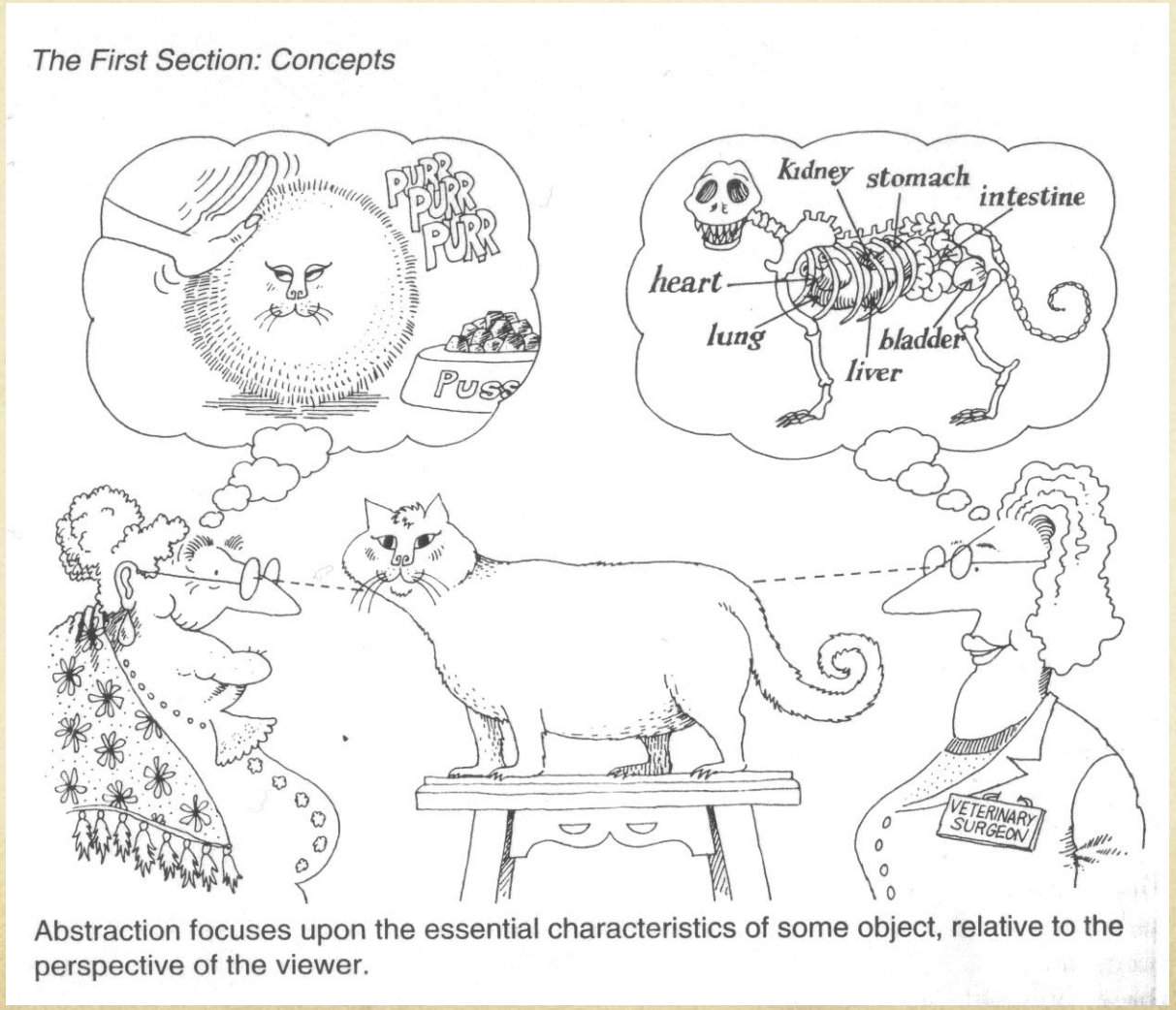
- An Abstract Data Type is a mathematical model defining the *functional* landscape that surrounds an entity (data item), implying:
  - The entity itself
  - The functions that act on that entity
  - Conditions under which the entity is operated on by the functions (preconditions)
  - Predicates (axioms) that specify the rules under which the functions operate
  - Finally, a class concept which encapsulates the entity and functional specification

# Abstract Data Types

- The conceptual boundary of an ADT separates the data and function from other related entities
- The ADT becomes the fundamental unit of decomposition in OO analysis
- The ADT is the first conceptual step toward modularization in OO analysis, but modularization is only fully *realized* by the OO notion of C*lass*
- A Class is a full or partial implementation of an ADT, *fleshing out* the functional landscape with a particular implementation of the ADT's semantics

# Abstraction

(from Booch, Object Oriented Analysis and Design, Benjamin Cummings, 1994)



# Abstraction

- Abstraction is a conscious and purposeful simplification of some aspect of the problem domain, allowing you to concentrate on the bigger picture in lieu of the myriad details (sc. Complexity)
- Implemented, Abstraction *exports commonality* into a parent or base class
- Encapsulation is in the end a form of abstraction

# Abstraction

- Risks:
  - Overabstraction: too many classes
    - too many classes for the same essential concept
  - Underabstraction: too few classes
    - the same code is in multiple places throughout the system
- Downsides:
  - Abstraction can complicate what Richard Gabriel calls code “habitability”: maintainers often find it hard to comprehend abstraction-laden designs
  - Over-abstraction tends to foster incorrect assumptions about a class’s semantics

# Downsides of Abstraction

- Abstraction can complicate what Richard Gabriel calls code “habitability”: maintainers often find it hard to comprehend abstraction-laden designs
  - Especially true in highly Polymorphic code
- Over-abstraction tends to guarantee incorrect assumptions about a class’s semantics

# Introduction to Inheritance

- Two key concepts will guide our thinking
- Inheritance supports two fundamental but related functions:
  - Allowing one *class* to automatically *reuse* the interface and/or implementation of another class's semantics
  - Allowing for the *substitutability* of one related *object* for another, strictly on the taxonomic (*type*) relationship that exists between them (Liskov Substitution Principle (LSP))

# Taxonomic Inheritance

- Inheritance implements an "*is-a*" or "is a type of" relationship:
  - A man or woman *is an* animal
  - A bubble sort *is a type of* sorting algorithm
  - A driver's license *is a type of* official documentation
  - A salaried employee *is a type of* Employee
  - A manager *is a type of* salaried employee

# Interface/Implementation Inheritance

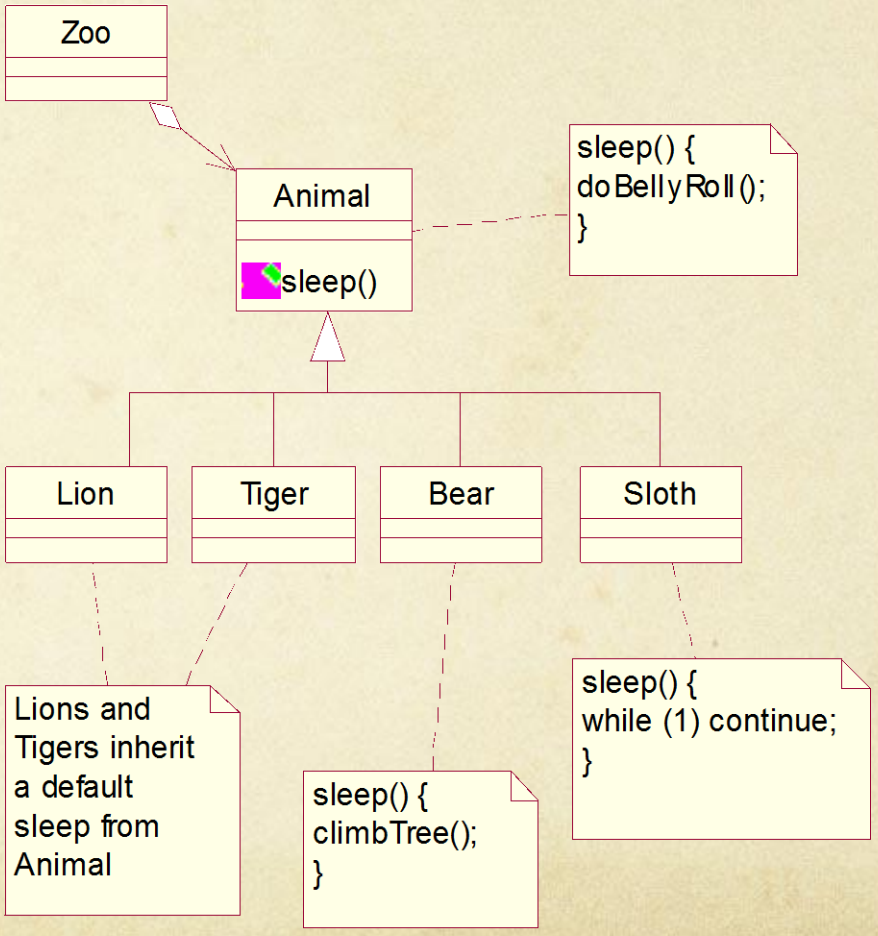
- When one class inherits the implementation of a set of operations from a parent (base, super) class, the code that realizes that implementation is *encapsulated* in the parent class, and allows:
  - All changes to that code to be localized in one place
  - All derived classes to be able to *reuse* that code without themselves having to provide their own implementation

# So What is Inheritance?

- Inheritance can be either singular or multiple:
  - A Checking Account *is a* type of Account
  - An instructor is an employee, a student is a person, and a teaching assistant is a type of instructor *as well as* a student. A teaching assistant *is both a* instructor *and a* student *at the same time*.

# Simple (Single) Inheritance

- A Zoo Simulation has animals (most development efforts are zoos)
- Individual Animals know how to sleep
- Abstract sleep into base class, so you can command *any Animal* to sleep
- *Implement* sleep() in concrete derivatives so each individual animal knows how to sleep

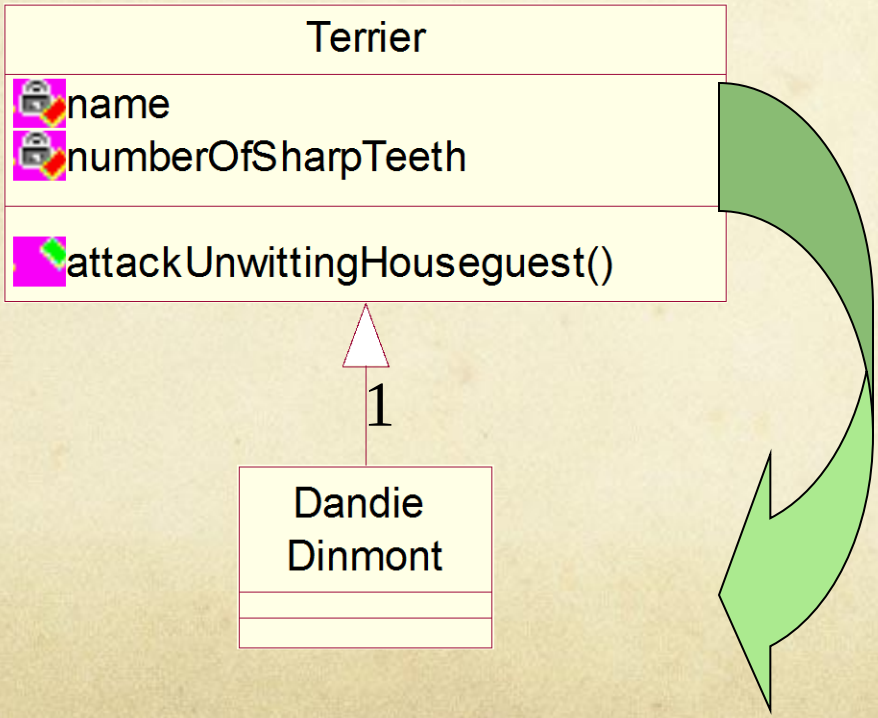


# Inheritance continued

- Inheritance is also often called a "Generalization/Specialization" relationship (Coad)
- Base class:  generalization
- Derived class:  specialization (it *extends* the capabilities of or further *specializes* the capabilities of the base class)
- Canine (generalization)
  - Collie (specialization)
  - Shetland Sheepdog (specialization)
  - Terrier (generalization and specialization)
    - Boston, Bull, Fox, Scottish, Clydesdale, Dandie Dinmont

# Single Inheritance

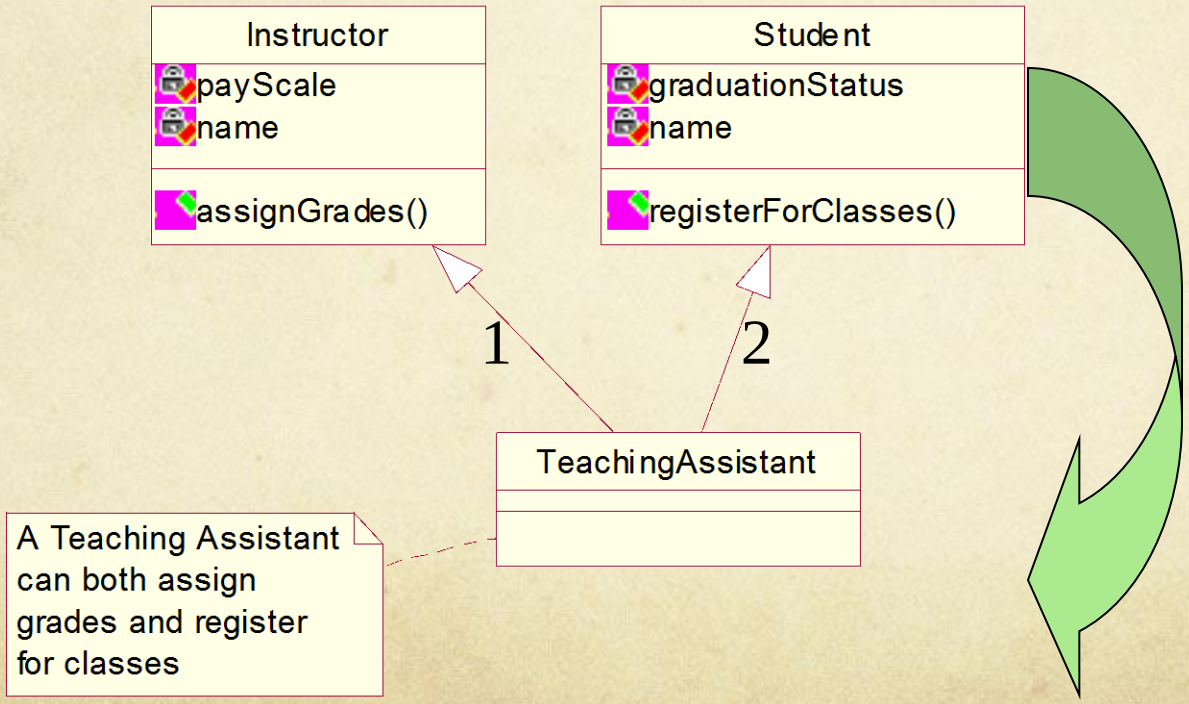
- Single Inheritance allows the inheritance of the implementation and/or interface of only *one* (single) base class




Implementation Inheritance provides the implementation as well as the interface to derivatives

# Multiple Inheritance

- Multiple Inheritance allows the inheritance of the implementation and/or interface of *multiple* base classes




TeachingAssistant inherits *both* interfaces of Student *and* Instructor, and, optionally, *both* their implementations

# Single vs. Multiple Inheritance

- Single:  Simula, Java, Smalltalk, Ada 95, Objective C, Ruby*
- Multiple: C++, Eiffel, Lisp (depth first), Python, Scala
- Implementation vs. Interface:  Java, C#
- Aspect Orientation*:  Ruby (mixins)
- Examples (fair or foul?):
  - Airplane + Corporate Asset → Company_Plane
  - Boat + Plane → Seaplane
  - Car + Person → Car_Owner
  - Vehicle + House → Mobile_Home
  - Tax Exemption + Infant → Well_Loved_Baby

# Multiple Implementation Inheritance: Good or Evil?

- Java, C#, Smalltalk, Ruby—they left it out.
- We should not be afraid of the variety of ways in which we can use inheritance. Prohibiting multiple inheritance … achieves no other aim than to hurt ourselves. The mechanisms are there to help you: use them well, but use them.—Bertrand Meyer
- Multiple Inheritance is like a parachute. You don't often need it, but when you do, you really need it. –Booch
- The lack of multiple inheritance (as in Java or Smalltalk) often creates additional work (implementation of a multitude of methods)
- It is precisely because of its complexity that multiple inheritance can be useful. The world is a complex place, and multiple inheritance allows us greater latitude to model its richness.—Wirfs-Brock

# Interface versus Implementation Inheritance

- Can you choose to inherit *just* the interface or do you have to inherit *both* the interface *and* the implementation?
  - C++: Yes (class and Abstract Base Classes)
  - Java: Yes (class via extends + interface implementation)
  - C#:  Yes (class and interface implementation)
  - Smalltalk, Eiffel:  No (implementation only)
  - Ruby:  No (aspectival mixin of implementation only)
  - Lisp:  Yes (metaobject protocol)

# Inheritance and Reuse

- Inheritance is one key avenue to deliver on the OO promise of reuse.
- Inheritance reuses types.

# Bertrand Meyer's Categories from *Object Oriented Software Construction*

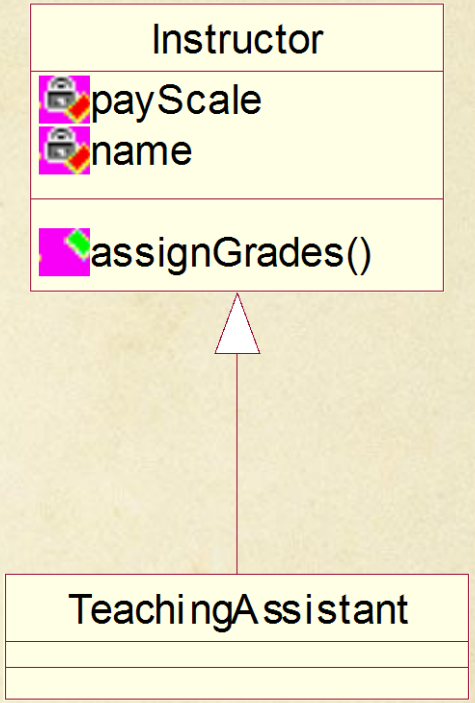
- Model Inheritance
  - *Subtype*: vertebrate <== mammal (most familiar form)
  - *Restriction*: rectangle <== square (constraint: 4 sides equal)
  - *Extension*: car <== Formula 1 Racer (extension (Java))
- Variation Inheritance (overloading/inherit to redefine)
  - *Functional*:  only method implementation modified
  - *Type*: both implementation and signatures are modified
  - *Uneffecting*: inherit an overly-concrete class and generalize its interface (better accomplished through delegatory facades)

# Heuristics for Determining Legitimate Inheritance

- Arthur Riel's Two Question Approach:
  - Q1: Is A a *type* of B?
  - Q2: Is B a *part of* A?
    - If Yes/No: Legitimate Inheritance
    - If No/Yes: Illegitimate Inheritance
  - Example: Should Airplane derive from Wing, Fuselage, etc.?
  - Example: Should a Collie derive from a Dog?
- Only subclass for taxonomic reasons, never for reasons of simple utility

# Heuristics cont.

- Subtypes specialize or extend.
- Within a given Problem Domain, the subtype should never need to morph into some other type of domain object. TeachingAssistant is a type of Instructor. Won't a TA ever need to be a Student (i.e., register for courses?)
- Subclasses must extend rather than nullify the Superclass (rules out Meyer's Uneffecting Inheritance)
- Do not subclass for simple utility (must subclass for taxonomic reasons, not simply for convenience)



# Difficulties in Inheritance

- MI and The Dreaded Directed Acyclic Graph (DAG)
- Does inheritance break encapsulation?
  - Versioning and the Fragile Base Class problem (Gene Hackman in *The Quick and the Dead*): The "rippling effect" of changing the base class (the primary condition of reuse)
  - Inheritance and "protected" variables: Possible side effects and by the way protected from whom?
- Back to simulation and imitation: inheritance allows a design to mimic the real world, but: to what degree is the world *really* hierarchical? And to what degree to we like good little Kantians impose a hierarchy on poor, defenseless business processes?

# Abstract Classes

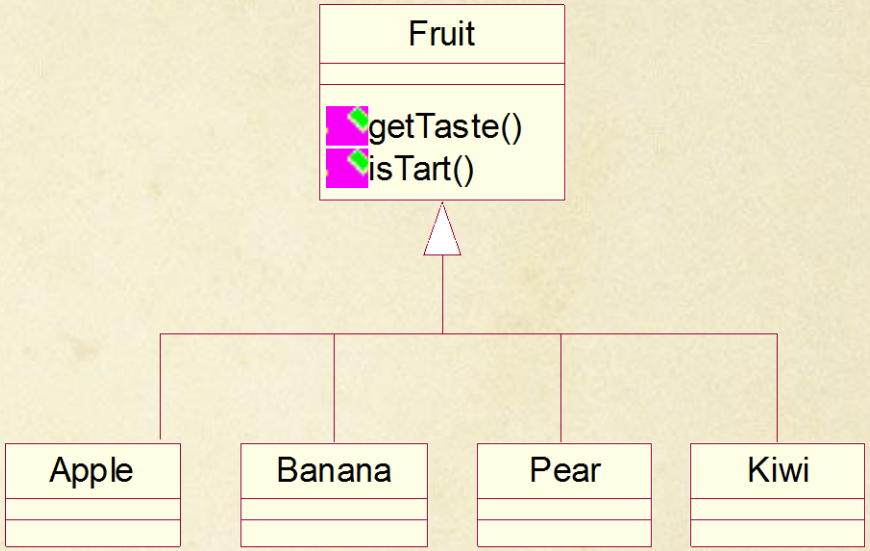
- Abstract Classes are types that cannot be instantiated, but represent an abstraction from other classes and serve to encapsulate common class interfaces and (perhaps) behavior
- Apples, Oranges, Pears, and Avocados are types of Fruit:
  - But what does “a fruit” taste like?
  - How many calories are in “a fruit”?
  - How much does “a fruit” cost?
- Waiter: What you like for dinner, Sir?
  - Customer: I’d like an appetizer, an entrée, and a dessert.

# Abstract Classes

- Shape s;

meaningless—a “shapeless” shape

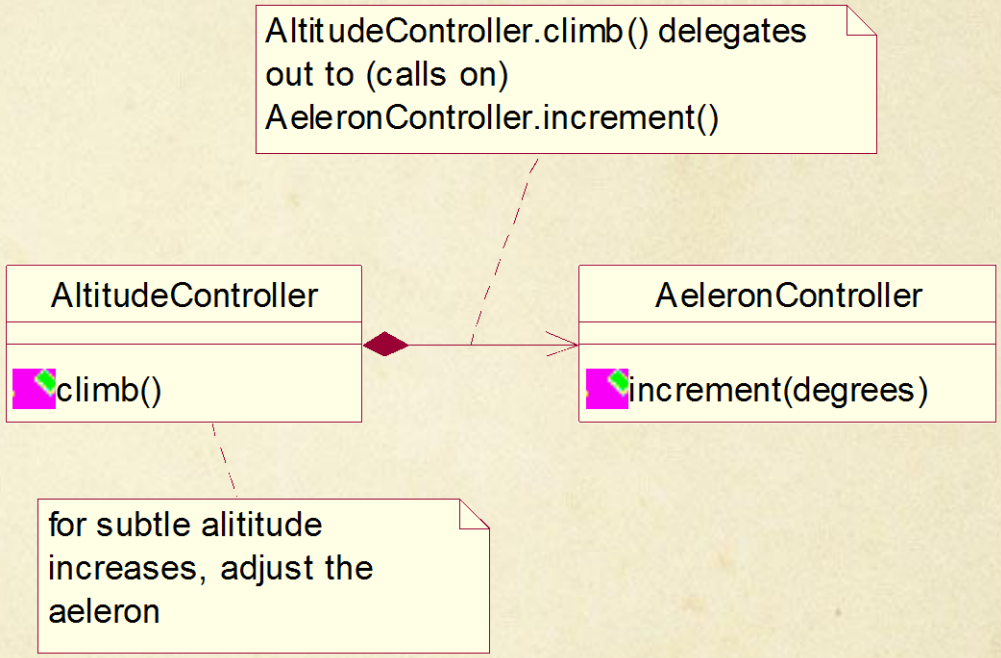
- An Abstract Class is:
  - a class that does not know how to instantiate itself
  - a class that therefore cannot be instantiated
  - a class that may include a partial implementation of its semantics (methods)



# Introduction to Composition

# Composition: Another Form of Reuse

- Composition is another form of *reuse*
- Instead of reusing capabilities by inheriting operations from a base class (which implies a taxonomic relationship), you embed another class within your class, and delegate out to (reuse) operations defined by the other class

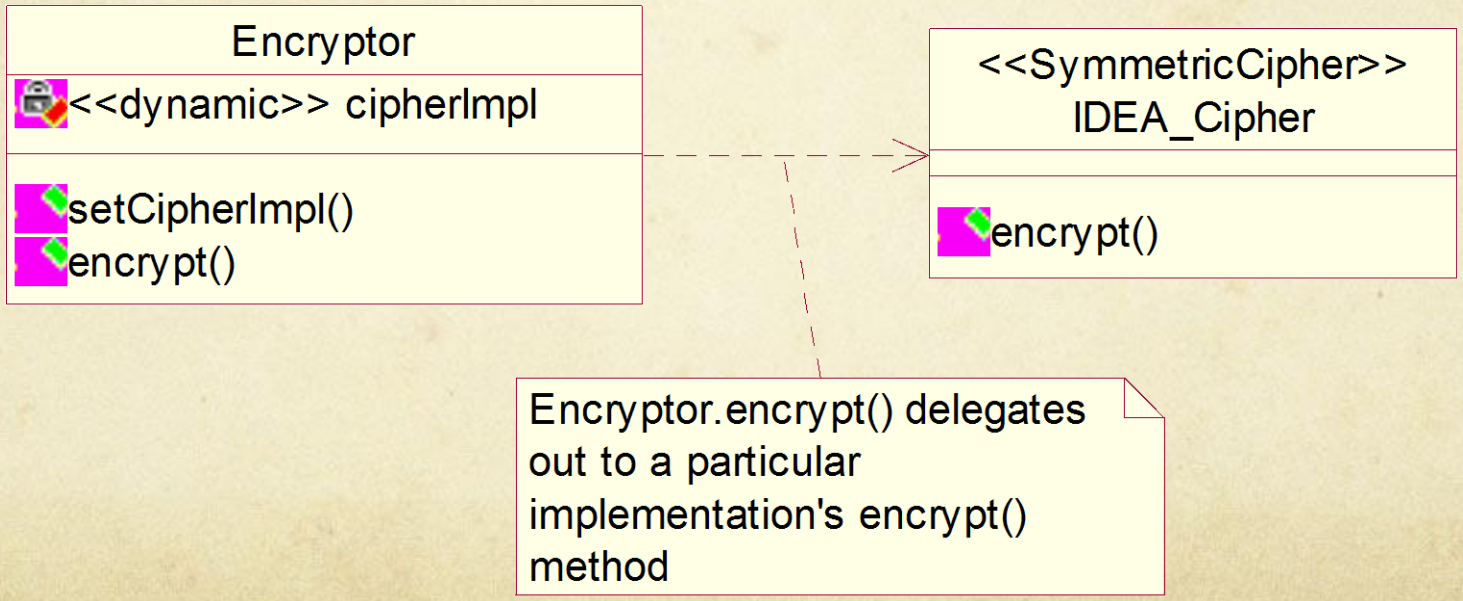


# Differences between Inheritance and Composition

- The composing class publishes the composed class's interface, and simply delegates out to the composed class for implementations
- Composition does not imply a taxonomic relationship between classes (an AeleronController is *not a type of* AltitudeController—*Oh*, *really*?)

# Dynamic Determination

- Composition is more flexible than inheritance because method implementations can be dynamically selectable (determined) at runtime



# Characteristics of Composition

- Composition models a *has-a* relationship, as opposed to Inheritance which models an is-a relationship
- Composition is a strengthened form of aggregation implying *simultaneous lifetimes*
  - Square and its four sides
  - Circle and its radius
  - hand and its fingers (robotics)
- Strong Composition implies:
  - a part cannot belong to more than one whole
  - concurrent instantiation and destruction with whole
- Cf. Robot class

# Why use Composition over Inheritance?

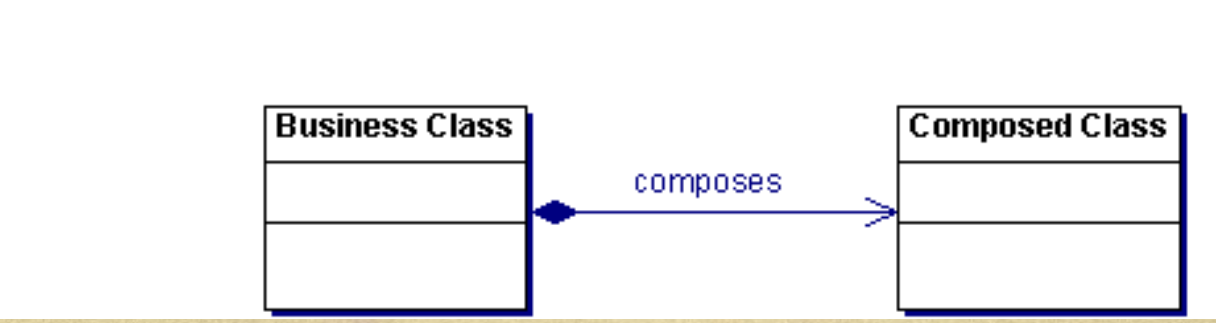
- Composition trades in reuse (through inheritance) for reuse through delegation (via reference), which allows for dynamic runtime configuration
- Use composition when you want to leverage (*reuse*) another class's *capabilities but not it's interface*
- Inheritance generally ties you to a particular interface *and* implementation
- This is limiting because:
  - you can't easily swap out that implementation for another at runtime
  - Inheritance hinders your ability to "design for change"

# Why Inheritance over Composition

- Inheritance makes global changes easier to make (change the base class, and eureka).
- Inheritance enforces type checking at compile time (in strongly typed languages)
- Delegation can complicate the reading of source code, especially in non-strongly typed languages (Smalltalk)

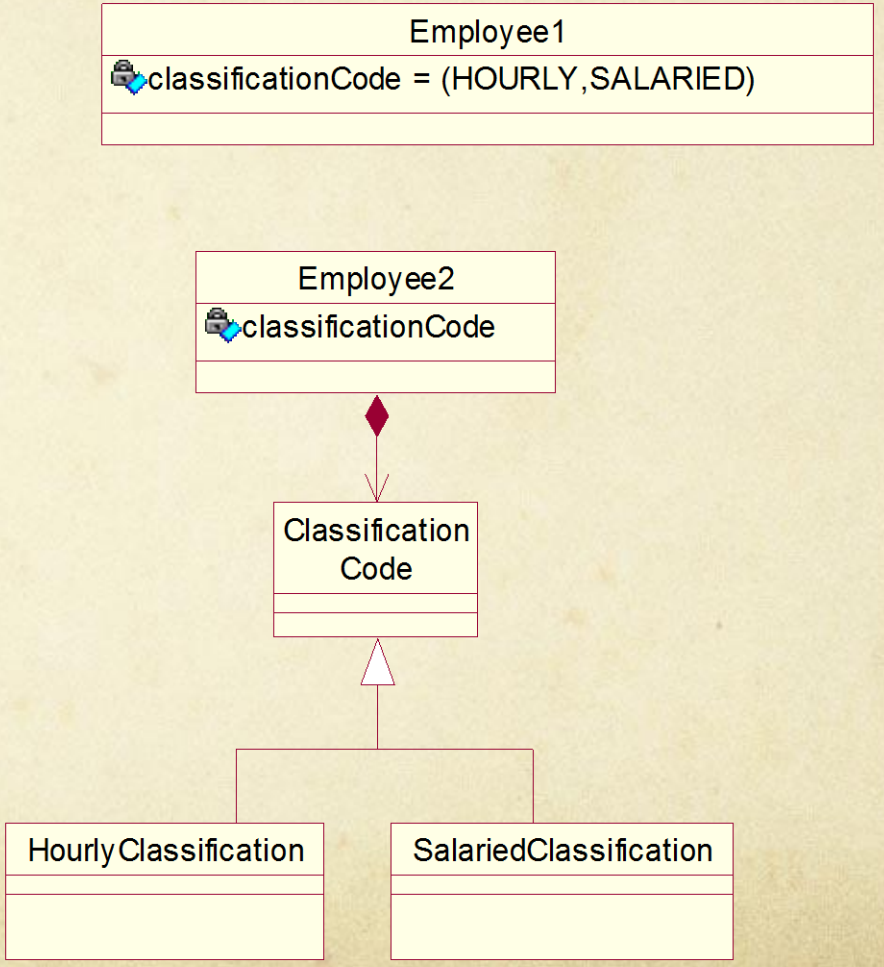
# Composition

- A stronger form of aggregation, which implies:
  - that the lifetimes of the whole and part are simultaneous (implicit construction/destruction of parts along with whole; cf. *constructor/destructor* role)
  - the nonshareability of parts belonging to a single whole (No my dear, that's *my* hand you're holding!)
- Longest lifespan relationship



# Composition and Attribution

- Attributes of a class can either be simple (*domain valued*) or relational (*delegatory*)
- An Employee may have a classification code: either hourly or salaried
- Which is "correct"?



# Heuristics

- Might classification codes themselves participate in a hierarchy? For instance, is an person earning commission a type of salaried employee? (=relational)
- Is the set of domain values predefined and not subject to change? Ie., regulatory mandated (=simple)
- Might you need to add a new classification code? (=relational)
- Would you ever need to individually manipulate (totalize, count, etc.) all salaried employees? (=relational)
- Do classification codes themselves have attributes (rates, etc.)? (=relational)

# Design Heuristics

- Prefer composition over inheritance except when:
  - a clear hierarchy of types exists *within the Problem Domain itself*, and this hierarchy is *never* expected to change
  - Defined roles exist that are never expected to change within a given Problem Domain

# Taxonomic Complexity

- *Kingdom*: Animalia
  - *Phylum*: Chordata (endoskeletal, closed circulation)
    - *Class*: Aves (birds)
      - Hey, Harry, are *Struthioniformes* (Ostrich) and *Raphidae cucullata* (Dodo) birds?
        - Not if all birds can fly.
- Problem Domain Abstraction and Reference
  - Such questions can only be answered in the context of a given *Problem Domain*
  - Model the ostrich flying problem through descendent hiding: fly() { null_op }

# Mimetic Complexity

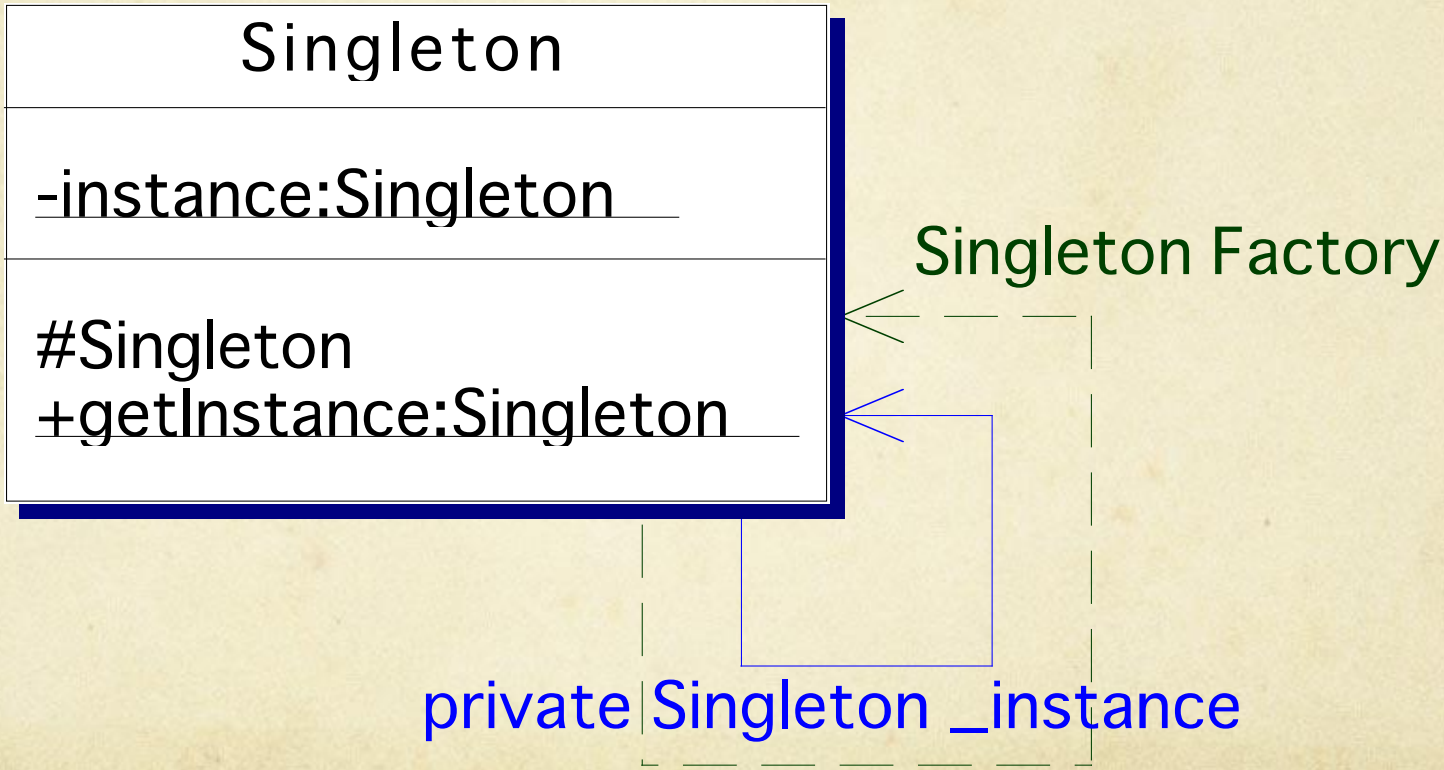
- It is subtly easy to confuse the exigencies of software design with the details of reality
- Suppose you are writing a matchmaking program for an online dating service:
  - Does Jim need to care who Sue's parents are?
  - Does Linda need to care about Bob's family tree, from whom he derives?
  - Does Larry need a unique identifier before he can introduce himself to Helen?
  - These are silly questions, they are about system design and convenience, not about love.

*An eclipse is something that happens between your eyes and the sun—not in the sun itself.*—Martin Buber

# Simple Pattern: Singleton

Ensure a class only has one instance, and provide a global point of access to it.

# UML


Singleton
-instance:Singleton
#Singleton
+getInstance:Singleton
Singleton Factory
private Singleton _instance

# Motivation

- Sometimes we need to ensure that there can only be one instance of an object within a given system
  - Filesystem acesss
  - Print Spooler
  - Thread manager
  - System Dictionary

# Strategy

- The Singleton works its magic by accomplishing two things simultaneously:
  - hiding the class's constructor from public view
    - this disables the ability for others to create new instances of the class
  - maintaining a single reference to a single instance of the class, internally in the class

# Benefits

- Controlled access to a single instance
- Enables an application to avoid the use of global references for single instances
- Provides a structure for the ability to have “Dual” or “Triad” objects (an object can control precisely two or three instances, and round-robin the incoming access to these objects
- Avoids the use of static class operations, which cannot be used polymorphically, because there is not object in play (no self or this pointer)

# Java Code (Simple)

```
public class Singleton {
private Singleton(){}
    public static Singleton getInstance(){
            if (instance == null) {
                instance = new Singleton();
            }
            return instance;
        }
private static Singleton instance = null;
}
```

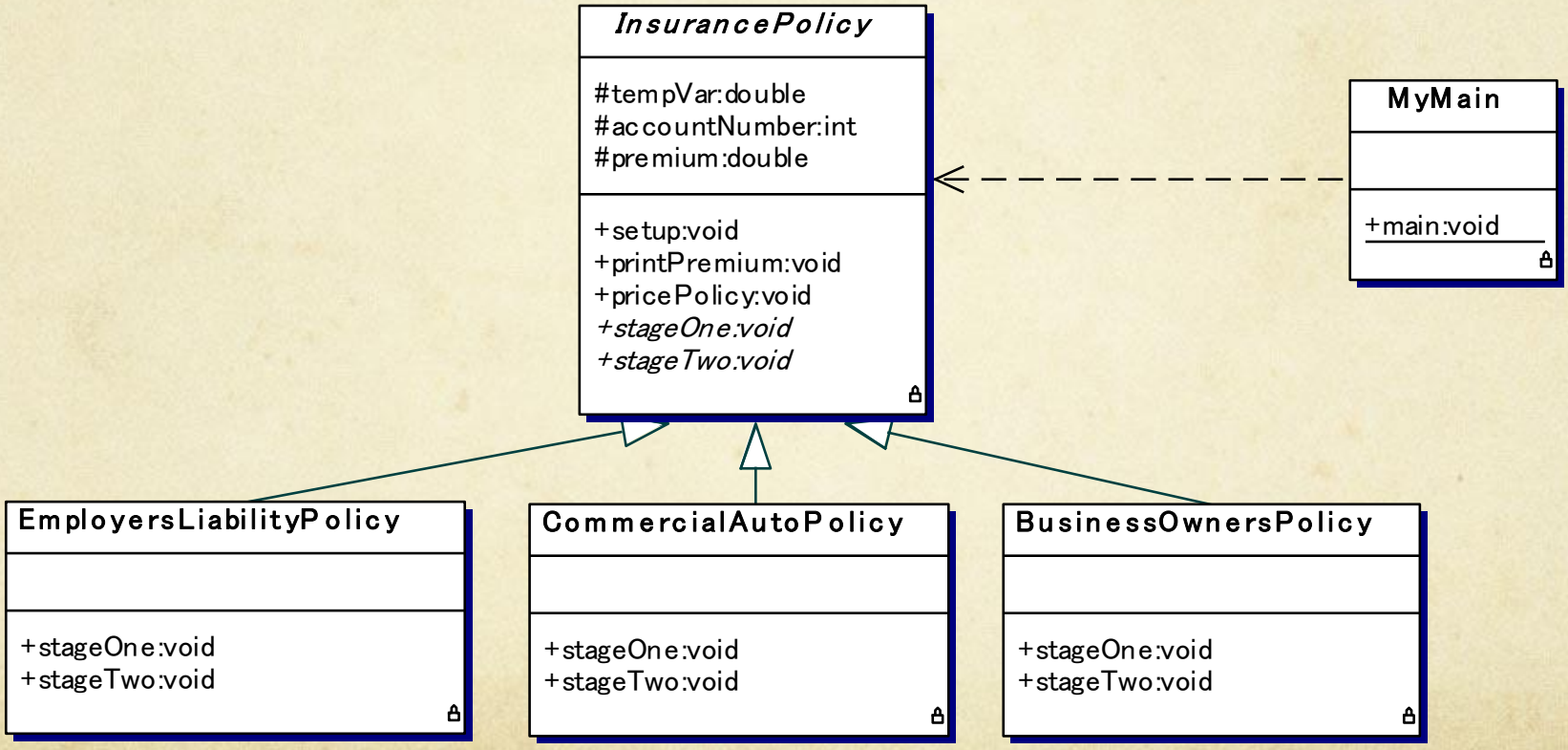
# Examples

- Java
- C++
- Smalltalk
- Ruby

# Template Method

Define the skeleton of an algorithm in an operation, deferring some steps to subclasses, letting subclasses define certain steps of an overall algorithm without changing the algorithm's structure

# UML


InsurancePolicy
#tempVar:double
#accountNumber:int
#premium:double
+setup:void
+printPremium:void
+pricePolicy:void
+stageOne:void
+stageTwo:void
MyMain
+main:void
EmployersLiabilityPolicy
+stageOne:void
+stageTwo:void
CommercialAutoPolicy
+stageOne:void
+stageTwo:void
BusinessOwnersPolicy
+stageOne:void
+stageTwo:void

# Motivation

- Sometimes, an algorithm is generic among multiple subtypes, with just a few exceptions
- These exceptions prevent the abstraction of the algorithm into a common base class
- Use the Template Method pattern to allow the invariant parts of the algorithm to exist in the common base class, and depend on derivatives to tailor the changeable parts of the otherwise common algorithm

# Benefits

- Because much of the algorithm can be encapsulated in an abstract base class, the invariant parts of the algorithm are localized and non-redundant
- Individual behaviors are easily accomplished and themselves encapsulated in derivative classes
- Predefined “hook” callback methods can be assigned at defined points in a sequence of activities

# TemplateMethod Example: Insurance Policy Rating Engine

- Java (MyMain)
- C++
- C#
- Smalltalk
- Lisp:  Common Lisp Object System (CLOS)

# Polymorphism

## The Dynamic Model:

## Behavioral Aspects

"Well," says Buck, "a feud is this way. A man has a quarrel with another man, and kills him; then that other man's brother kills *him*; then the other brothers, on both sides, goes for one another; then the *cousins* chip in—and by and by everybody's killed off, and there ain't no more feud. But it's kind of slow, and takes a long time. . . . ."—*Adventures of Huckleberry Finn*

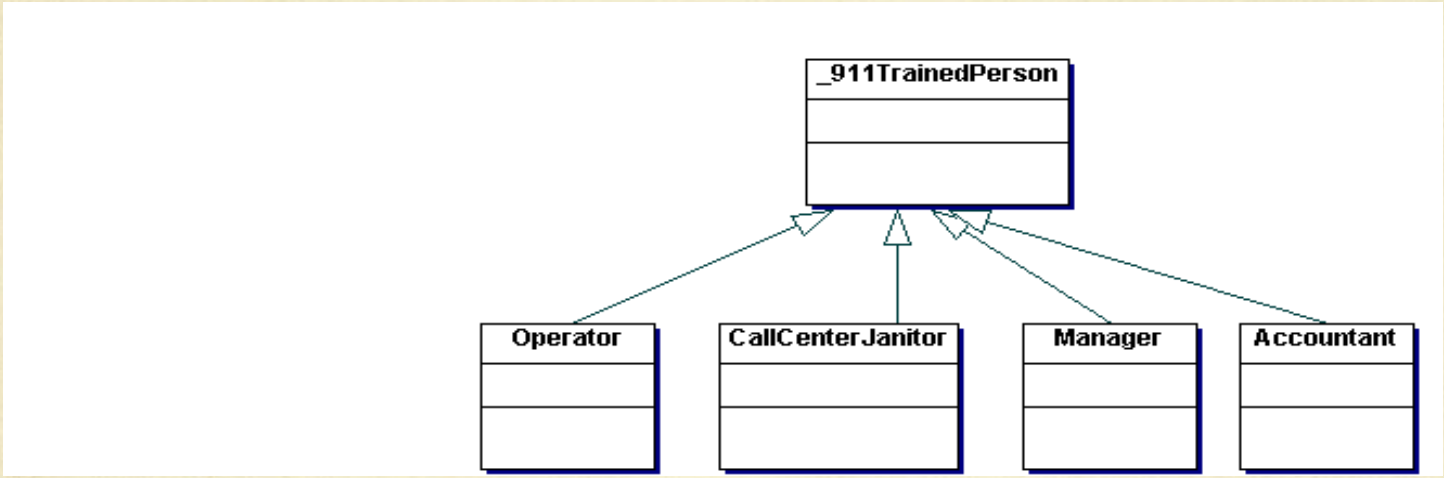
# Polymorphism

- Examples of the concept:
  - Huckleberry Finn and the Grangerfords and Sheperdsons
  - Bluto (Belushi) and Otter in Animal House: Delta house versus Omega House, after being beat up, Bluto says: “What … happened to the Delta I used to know? Where's the spirit? Where's the guts, huh? Ooh, we're afraid to go with you Bluto, we might get in trouble… Not me! Not me! I'm not gonna take this. Wormer, he's a dead man! Marmalard, dead! Niedermeyer, dead, Greg, dead…”
  - A fire call to whoever is at the 911 departmental desk: Fire Chief, Lieutenant, Sergeant, Office, Maid, etc.

# Polymorphism

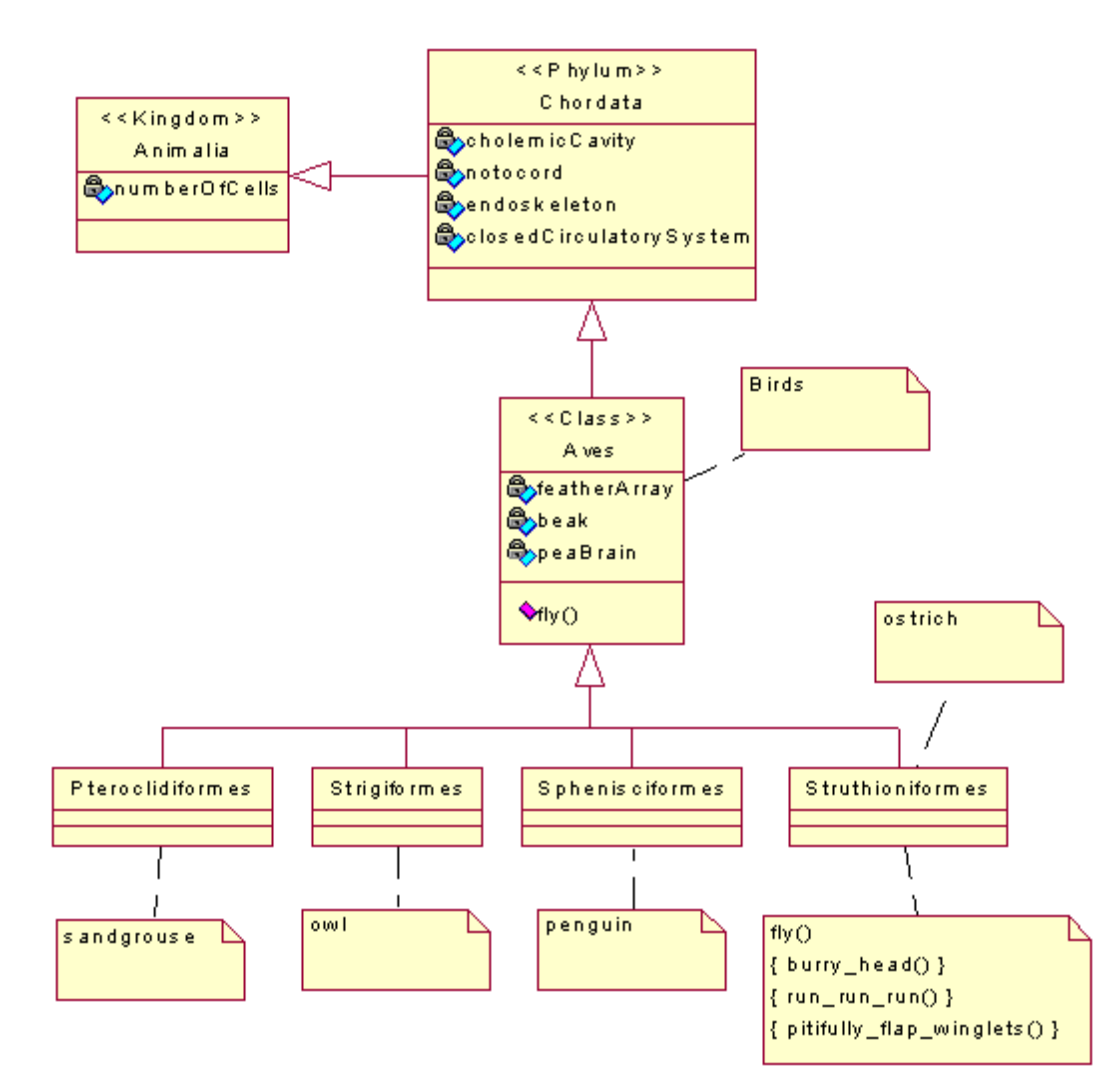
- Polymorphism simply means that you can *command* an instance of a *subtype (some type of thing)* by issuing a command on the base class interface without having to know or care about the *specific subclass type*
- Polymorphism is therefore not politically-correct (it completely removes concern for individuality)
  - Private!  Pick up that M60 and head up that hill!

# So what does this mean?



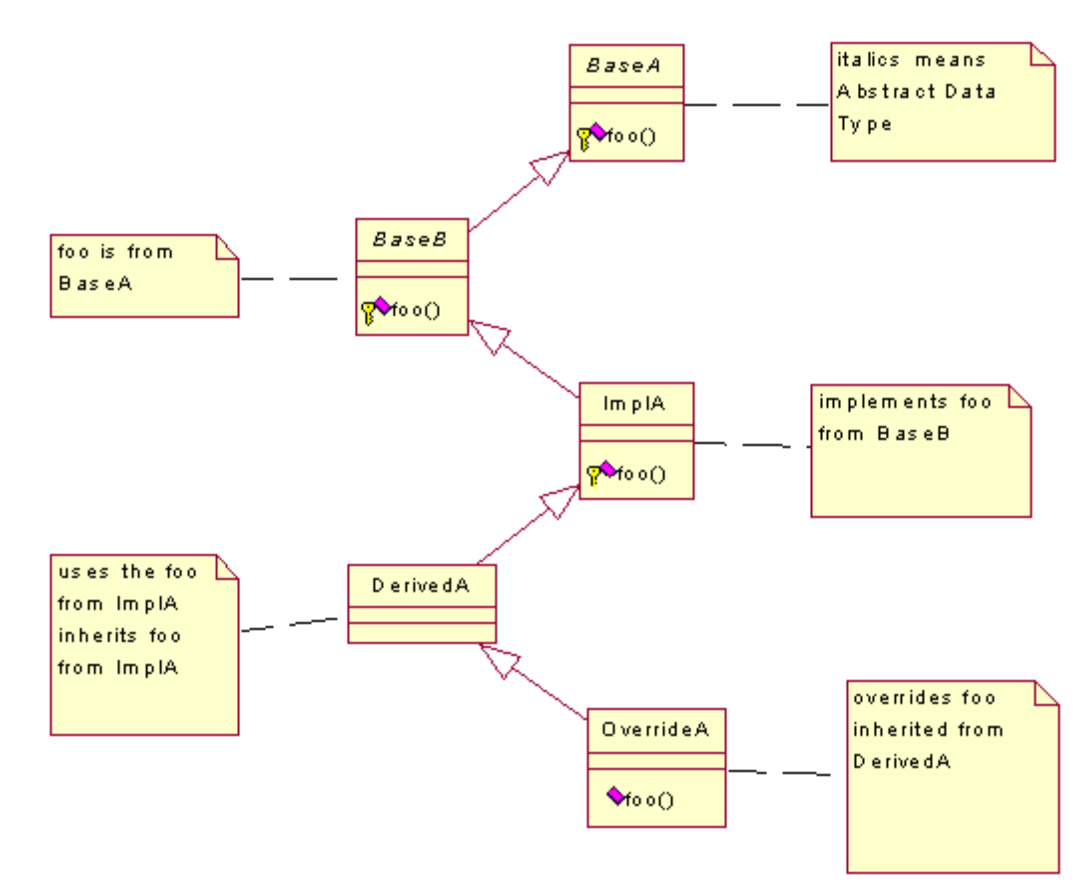

- *Every* employee of the 911 Call Center knows how to handle a 911 call, *regardless of "who they are"* in the organization

# Polymorphism Revisited



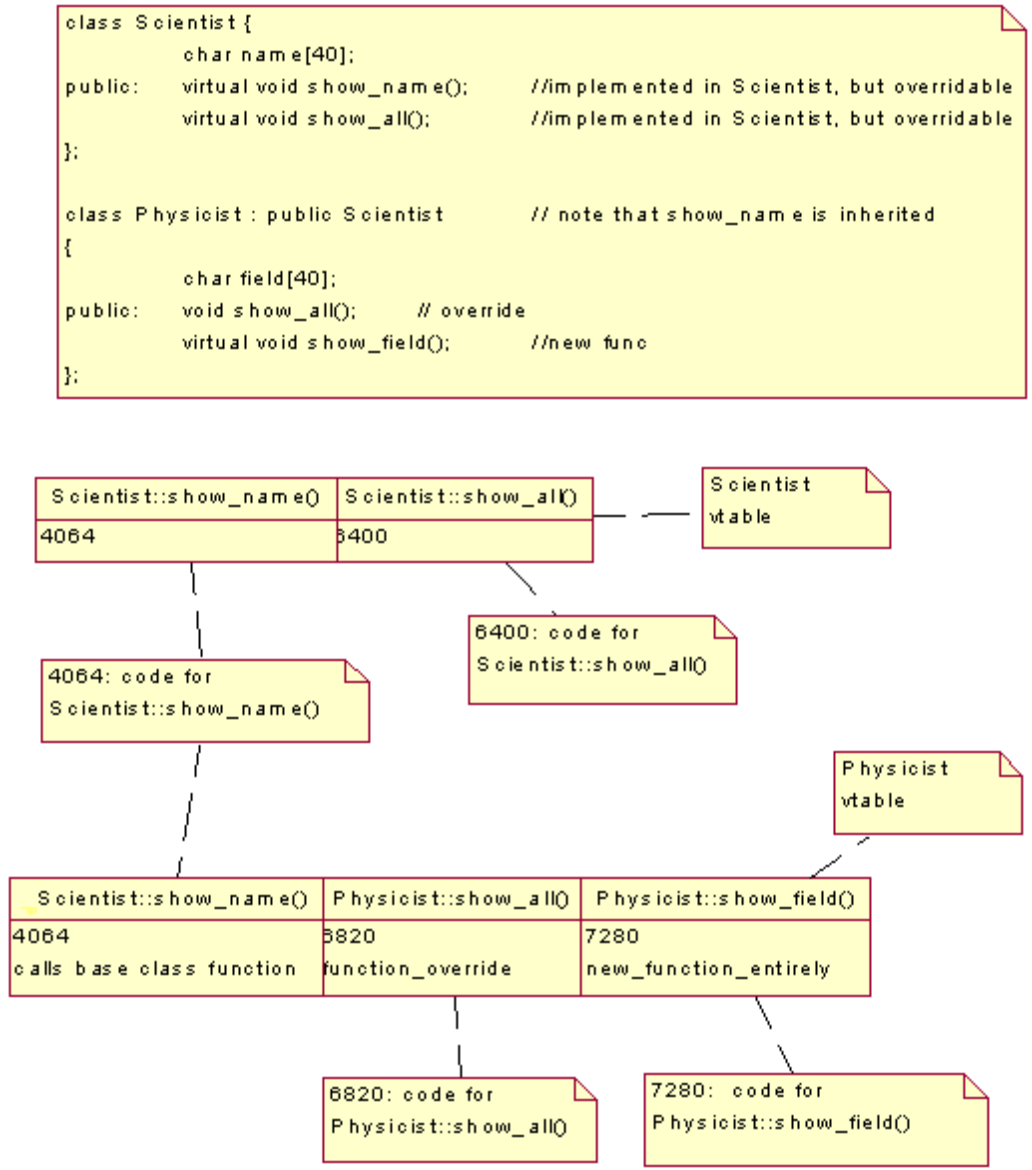

- $(\forall x)(Bx \rightarrow Fx)$
- All Birds Fly
- An Osterich is a Bird
- An Osterich flies?

# Essential Polymorphism




- A client has a reference to BaseA, instance could be BaseA, ImplA, DerivedA, or OverrideA
- A client has a reference to ImplA, instance could be ImplA, DerivedA, or OverrideA
- A client has a reference to DerivedA, instance could be DerivedA, or OverrideA
- A client has a reference to OverrideA, instance could only be OverrideA
- ***Example***: notvirt in Java and C++

# Fast Poly, C++ style




- Physicist p = new Physicist(“Feynman”);
- Scientist * s = &p;
- s->show_all();
- s->show_name();
- ((Physcist)s) ->show_field();
- ***Example*: scientist.cpp**

# Typing

- Strongly-typed languages ensure each variable has an defined type, and can only reference objects at runtime that belong to that type (or its derivatives). The compiler guarantees that calls will not fail at runtime. (C++, Java, C#)
- Weakly-typed languages use variables that have no *inherent* type associated with them, variable names are *generic* object references, and can refer to *any* object whatever (Eg. Smalltalk, lisp)
- *Example*: myBagBaby in Smalltalk (music.ws)

# Binding

- Binding refers to both:
  - when a *type* is bound to a variable or
  - when a *method is dispatched* to an object:
- Strict Early *Type* Binding: C++ language
  - strictly *compile-time* binding
  - *Allegiance* to type…
  - int x; // better be an int or you're casting…

# Early Typing / Late Binding

- Compile-time Typing: Java (*run-time* binding)
  - Compiler determines which *method signature* to call at *compile time* based on type of parameters (“I’m going to call some object’s *put* method that takes in a *Collection* and an *integer*: ???.put(Collection, int)”)
  - Virtual Machine determines which *target object* to send the message to based on the actual object referred to *at runtime*.
  - *Example*:
    - Java: Music.java

# Late Method Binding

- Late Binding: Smalltalk (*run-time* binding)
  - No *allegiance* to type in variables
  - Two classes, Rectangle and Inventory, both define a method called *size*
    - MyRef := Rectangle new.
    - MyRef := Inventory new.
    - MyRef size. “to which object is *size* dispatched?”
    - What happened to the Rectangle?
    - In Smalltalk, a method is dispatched at runtime according to the *method name* (selector) and the parameter order of its *arguments*
    - *Example*: Smalltalk: Symphony class

# Dynamic Typing Revisited: Duck Typing

- If it walks like a duck, quacks like a duck, and has webbed feet, it's a duck (cf. Justice Potter Stewart, in Jacobellis v. Ohio (1964), James Whitcomb Riley, the Hoosier Poet)
- DT is a dynamic typing in which an object's methods and properties determine the valid semantics, rather than its type of class or implementation of a specific interface
  - C# (~~obj is MyClass~~) or Java (~~instanceof(MyClass)~~)
- Therefore, DT focuses on *capabilities* rather than type, and these capabilities are discovered at runtime (dynamically)
- Smalltalk legacy (mybagbaby), C# (4.0), Python, Ruby, Lisp, Scala
- Downsides (Little Red Riding Hood Syndrome)

# Binding and Polymorphism

- Hybrid (C++)
  - C++ with virtual function tables (each object has a vptr in the class's vtable)
  - C++ virtual function polymorphism works only with pointers and references, not composed objects
  - Fast, no hashing is done, polymorphic behavior is determined by direct vtable offsets
  - pointers and references, not objects.
  - *Example*: music.cpp

# Hybrid Languages

- Some languages offer both early and late binding, at the *direction* of the programmer (C++, C#, Eiffel)
- When used polymorphically, the compiler determines which method to call based on *both* the *operation name and* the *static types* of the *parameters*
- *Example:* Music.cs (new versus virtual/override)

# Multiple Dispatch (Multi-Methods)

- Some languages not only do not type variables, but methods do not *belong to* classes *per se*, but are generically defined in the environment
- MD languages decide on an implementation of a method ***only*** *at runtime*, based on the actual *types* of parameters of a runtime call
- Thus, depending on the *types* of parameters *discovered* at runtime, the best method *to apply to an object will be selected* from any number of relevant choices
- Thus, a lisp class *has* attributes (slots) but methods are *applied* to a lisp object *at runtime*
- In such languages, the notion of object *self* is missing
- *Examples*: Lisp: music.cl

# Binding and Polymorphism

- Pure Polymorphism (Java Interface, C++ ABC)
  - interface vs. implementation inheritance
  - a pure polymorphic method is one which does not have a base implementation
  - a pure virtual function (or an ABC in C++) provides a "placeholder" method name, but does not provide an implementation because the method is *generically* meaningless
  - a "taste" method on a fruit abstract class...

# Covariance and Contravariance

- Covariance implies converting from a specialized type (Dandy Dinmont) to a more general type (Terrier): Every Dandy Dinmont is a terrier. [Broadening]
  - Covariance = Specialized –> General (think *up*)
- Contravariance implies converting from a general type (Shapes) to a more specialized type (Rectangle). [Narrowing]
  - Contravariance = General –> Specialized (think *down*)
- C# Delegates support Covariance and Contravariance
  - Delegate methods can vary (narrow) on inherited return types
  - Delegate methods can vary (widen) on inherited parameterized types

# Covariance and Contravariance

- Example from C# 4.0:
  - Let: `delegate object MyCallback(FileStream s);`
  - Covariance makes the following legal (return type):

```
string SomeMethod(FileStream s);

//both strings and FileStreams are Objects
```

  - Contravariance makes the following legal (parameter type):

```
Object SomeMethod(Stream s);

//Stream is a base class of FileStream
```

  - However, the following is illegal:

```
Int32 SomeMethod(FileStream s)
```

    - Because Int32 is a value type, not a reference type, and thus cannot participate polymorphically (autoboxing does not apply in delegates)

# Hallmarks of Good OO Design

- Short methods
- obvious methods (self-evident naming)
- no "God" objects (no one is omniscient)
- no "Manager" objects (nobody is too busy)
- trust your objects to "do the right thing"
  - handle errors through exception interface
- good objects have clear responsibilities
- TELL, don't ask.