Object-Oriented Programming

- Identifying classes and objects
- Types of class relationships
  - A uses B
  - A has-a B
  - A is-a/is B
- Inheritance relationships
- Polymorphism
- Reading:
  - Dawson, Chapter 9
  - http://introcs.cs.princeton.edu/python/33design/

Identifying Classes and Objects

- A class represents a group (“class”) of objects with the same attributes and behaviors
- Generally, classes representing objects should be given names that are singular nouns
- Examples: `Coin`, `Student`, `Message`
- A class represents the concept (or "blueprint") of such an object
- We are free to instantiate as many “instances” of each object as needed
- Good selection of object names for the instances can be helpful to understanding

Identifying Classes and Objects

- We want classes with the proper amount of detail - neither too much ***nor*** too little
- For example, it may be unnecessary to create separate classes for each type of appliance in a house
- It may be sufficient to define a more general `Appliance` class with appropriate instance data
- It all depends on the details of the problem being solved

Identifying Classes and Objects

- Part of identifying the classes we need is the process of ***assigning responsibilities*** to each class
- Every activity that a program must accomplish must be represented by one or more ***methods*** in one or more classes
- We generally use ***verbs*** for the names of methods
- In early stages it is not necessary to determine every method of every class – begin with ***primary*** responsibilities and ***evolve*** the design

Class Relationships

- Classes in a software system can have various types of ***relationships*** to each other
- Four of the most common relationships:
  - Dependency: A ***uses*** B
  - Aggregation: A ***has-a*** B (as in B is an integral part of A)
  - Interface: A ***is*** B (adjective) or A ***is-a*** B (noun)
  - Inheritance: A ***is-a*** B
- We will mainly focus on the first two and the last
- *interface* has different meanings...

Dependency

- A *dependency* exists when one class relies on another in some way, usually by invoking the methods of the other
- We've seen dependencies in previous examples and in Projects 1 and 2
- We don't want numerous or complex dependencies among classes
- Nor do we want complex classes that don't depend on others
- A good design strikes the right balance

Dependency

- For example, a <u>DicePlayer</u> object *uses* two <u>Die</u> objects – rolling them on each turn
- If we wrote software for a taxi service, we might have classes for <u>Driver</u> and <u>Taxi</u>
- The relationship between the two would be one of dependency. A <u>Driver</u> drives a <u>Taxi</u>
- Dependency indicates a relationship where one type <u>*uses*</u> the other – but neither is considered part of the other.
- We say that <u>**A**</u> "uses" <u>**B**</u>

Aggregation

- One of the benefits of object-oriented programming is that we can define new types composed of other types
- An *aggregate* is an object that is made up of other objects
- Therefore aggregation is a *has-a* relationship
  - A `Car` *has a* `Transmission` and *has an* `Engine`
  - A `StudentBody` *has* several `Student` objects
  - A `CoffeeMaker` has a `Heater` and a `Container`
- These parts can be basic built-in types, or other custom-made types

Aggregation

- In code, an aggregate object contains references to its component objects as instance data
- The aggregate object itself is ***defined*** in part by the objects that make it up
- This is a special kind of dependency – the aggregate usually ***relies for its existence on the component objects***
- As we saw with the `Address` problem in class, it can be very useful to deal with the aggregate as a self-contained unit, rather than trying to juggle separate parts

Aggregation

- There are two ways to include the component objects in an object that is an aggregation
  - For one component (or a small constant number of components), use parameters in the constructor

```
def __init__ (self, first_name, last_name,
                  street,...):
```

  - For a large or indefinite number of components, you can create an empty list, along with a function to add items

```
def place_order (self, name, flavor, size):
              ...
```

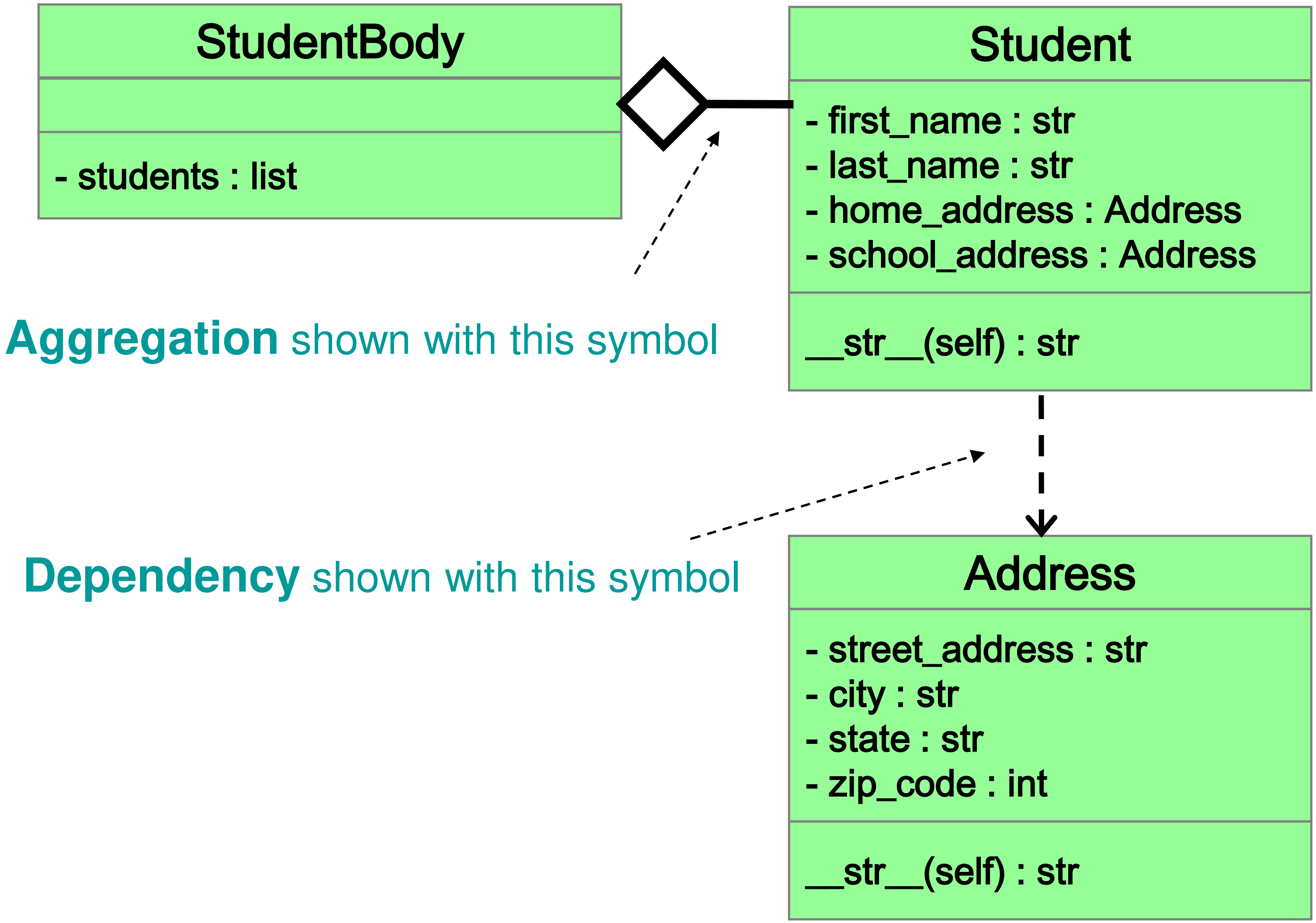
UML – A *Modeling* Standard

- UML is a graphical tool to visualize and analyze the requirements and do design of an object-oriented solution to a problem
  - Allows you to visualize the problem / solution
  - Organizes your detailed information
- We have seen this before, implicitly.
- It's a complex topic, but we will examine one part: *class diagrams*

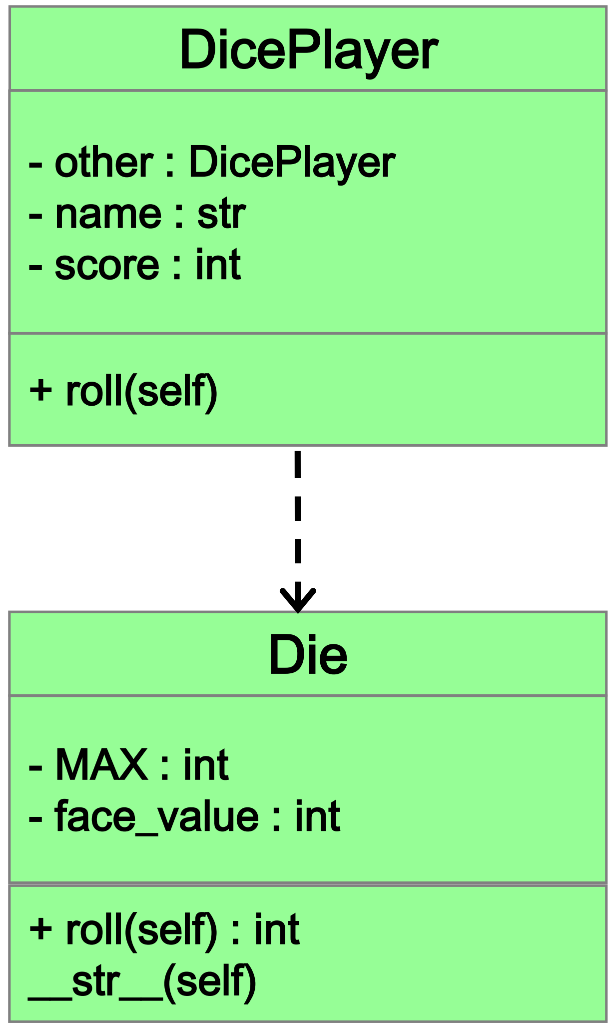
Class Diagrams

- Classify the object types of the program
- Define name of each class
- Define each class's members:
  - Attributes (variables)
  - Behaviors (functions)
- Show *relationships* between classes
  - Dependency
  - Aggregation
  - Inheritance

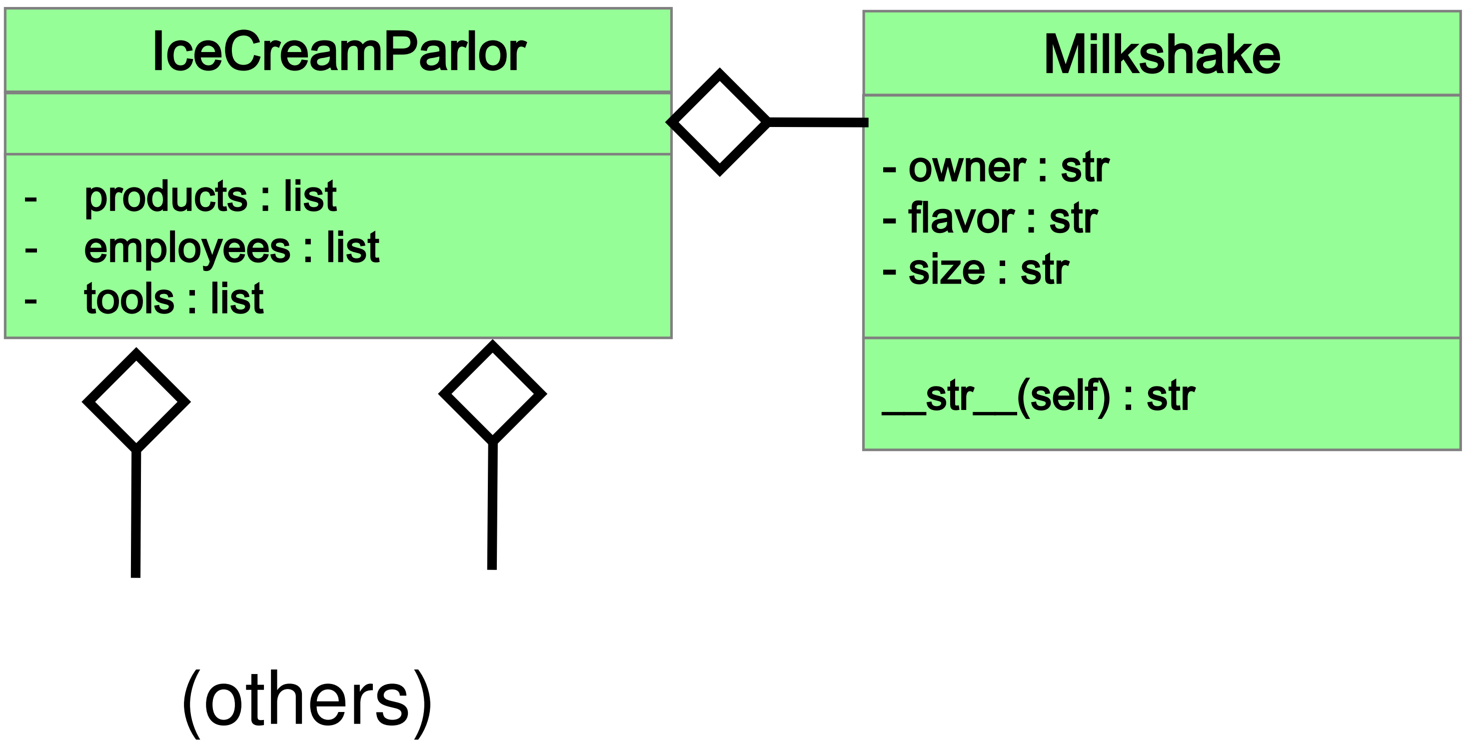
Dependency/Aggregation in UML



Dependency/Aggregation in UML



Dependency/<u>Aggregation</u> in UML



Inheritance

- *Inheritance* allows a software developer to derive a new class from an existing one
- The existing class is called the *parent class, superclass*, or *base class*
- The new class is called the *child class, subclass* or *derived class*
- As the name implies, the child inherits characteristics of the parent – i.e., its attributes and data

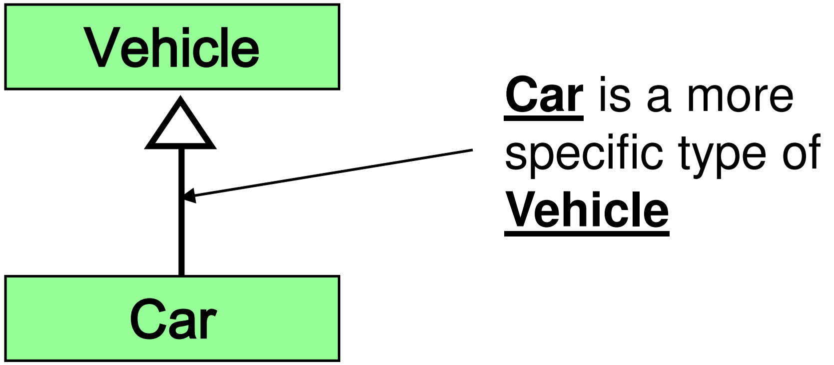
Inheritance

- *Software reuse* is a fundamental benefit of inheritance
- As they say, "Don't reinvent the wheel". Take advantage of what others have done well.
- A programmer can tailor a derived class as needed:
  - adding **new** variables and functions
  - **"overriding"** some of the inherited methods
- An inheritance relationship specifies that:

  A *is a* **B**

Inheritance

- Inheritance is based on an *is-a* relationship
- The child *is a* **more specific version of** the parent
- Inheritance relationships are shown in a UML class diagram using a solid arrow from the child class to the parent class



Deriving Subclasses

- In Python, we use the class header line to establish an inheritance relationship
- Specifically, we place the *parent* class name in parentheses after the class name.
- In the example below, we are creating a **Car** class that is based on a more general **Vehicle** class

```
class Car (Vehicle):
    # class contents
```

Overriding Members

- Left alone, a child class will inherit all the public functions of its parent class – as if you "copied and pasted" the code into this one.
- However, a child class can redefine (or *"override"*) the definition of a public inherited function in favor of its own, more class-appropriate, definition
- In Python, code is interpreted and executed at runtime, so this is where types come into play.
- The class of the object determines which version of the function is invoked at execution
- If you have a public variable in the parent class, and then attempt to assign a variable of the same name in the child class, it will overwrite the previous value.

The *super* Function

- In Python, constructors and other public functions are inherited from the parent class.
- Yet we often want to use the parent's version of the function *inside* the child's version
- The `super` function can be used to refer to the parent and invoke the parent's version

```
def Child (Parent):

    def __init__(self):
        super().__init__()    # a call to Parent()
        # plus whatever code we need for Child
```

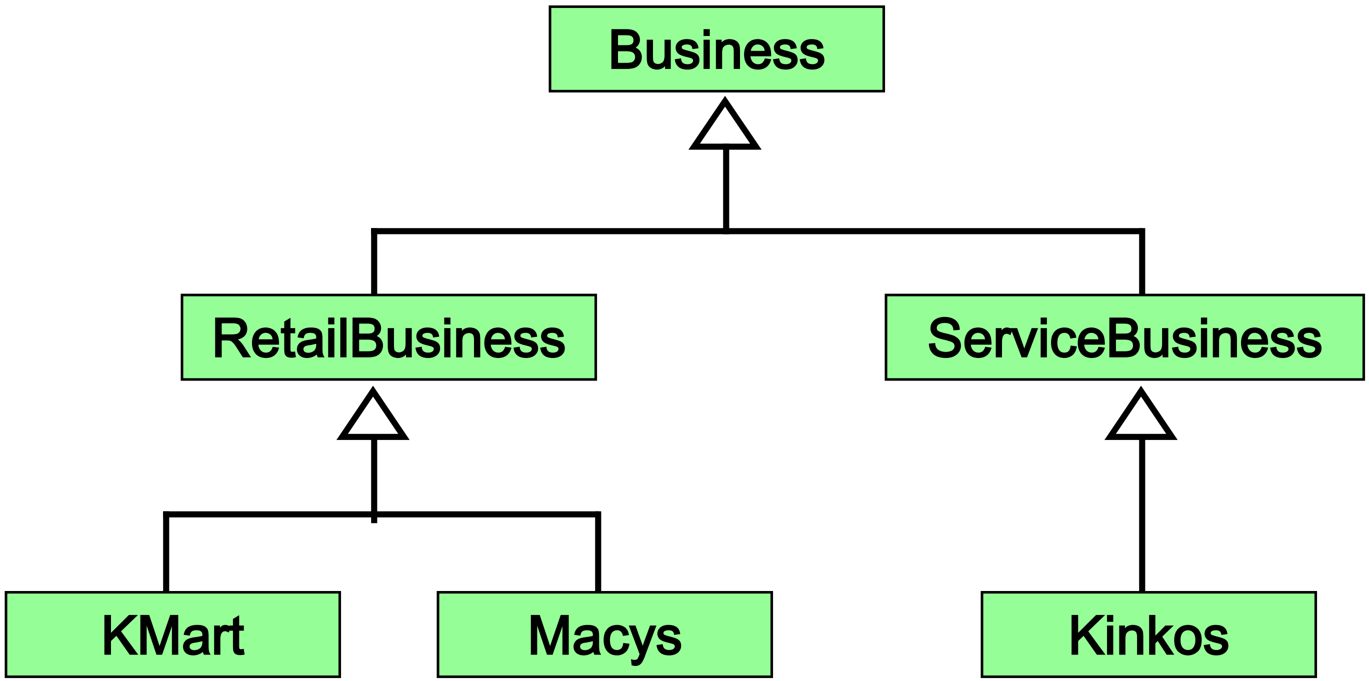
The super Function

- The **<u>first</u>** line of a child's constructor should use the `super` reference to call the parent's constructor
- The `super` reference can also be used to reference (with a dot **<u>.</u>**) other variables and methods defined in the parent's class:

```
def my_function (self):
        super().my_function()    # a call to the
                                 # parent version
        # plus whatever code we need for child
        # class version
```

Class Hierarchies

- A child class of one parent can be the parent of another child, forming a *class hierarchy*
- Two children of the same parent are called *siblings*



Class Hierarchies

- A child class inherits from **all** its ancestor classes
- An inherited variable or function is **passed continually down the line** (unless it is declared *private*)
- Common features should be put **as high in the hierarchy as is reasonable**
- There is **no** single class hierarchy that is appropriate for all situations

Interface

- Here, we get into the notion of an *interface*
- In some languages, such as Java, there will be a formal structure called an interface
- (In fact, in Java, "interface" is a reserved word!)
- At the very least, though, the term refers to the idea that there are certain *operations* and *messages* you can apply to something
- For example, you can:
  - ➢ Perform ***arithmetic*** with numbers
  - ➢ Access ***element*** and ***slices*** of sequences
  - ➢ Fetch ***values*** from dictionaries using ***keys***

Interface

- Sometimes, completely different types may share a *similar interface*
- For example, the **+** operator can be applied to both numbers and strings
- For sequences and dictionaries, you can also use the same syntax for getting elements:

  `variable[index_or_key]`
- In Project 2, you may notice that the different shape classes share in common an area function, so you can call .area() for any of them!
- This makes it possible to loop through a sequence of different shape objects without having to change the code for any of those types.
- These are examples of **polymorphism**.

Polymorphism

- The term *polymorphism* literally means "having many forms"
- *Polymorphism* is in effect when we can treat *different types* in a *similar manner*
- Many operations and function calls in Python are potentially polymorphic
- You can apply similar actions towards very different types, such as adding numbers vs. concatenating strings.

Designing for Inheritance

- As we've discussed, taking the time to create a good software design reaps long-term benefits
- Inheritance issues are an important part of an object-oriented design
- Properly designed inheritance relationships can contribute greatly to the elegance, maintainability, and reuse of the software
- Let's summarize some of the issues regarding inheritance that relate to a good software design

Inheritance Design Issues

- Every derivation should be an is-a relationship
- Think about *a potential future class hierarchy*
- Design classes to be *reusable and flexible*
- Find *common characteristics* of classes and push them as high in the class hierarchy as appropriate, i.e. “generalize” the behavior
- Override methods as appropriate to tailor or change the functionality of a child
- Add new variables to children, but only redefine inherited variables if you mean it

Inheritance Design Issues

- Allow each class to manage *its own data*; use the `super` reference to invoke the parent's constructor to set up its data
- Even if there are no current uses for them, override general methods such as **`__str__`** and **`__eq__`** with appropriate definitions
- You can use *super classes* to represent general concepts that lower classes have in common
- Use *visibility modifiers* carefully to provide needed access without violating encapsulation