Intro to Computer Science

Previous

- Objects (review)

Next

- Advanced OOP
  - Inheritance
  - Encapsulation
  - Polymorphism

Readings	
Gaddis	• Chapter 10

Readings	
Gaddis	• Chapter 11

Benefits of OOP

1. Encapsulate data and functionality into a single structure
   - Users don't worry about details
   - Developers are free to change their mind about details
2. Program with abstract concepts
   - Focus is on types
3. Make the job of a developer "easier"
   - Work with types rather than values
   - Build on others work

Benefits of OOP

1. Encapsulate data and functionality into a single structure
   - Users don't worry about details
   - Developers are free to change their mind about details

Encapsulation

- Keep data, and the code that operates on that data, in a single place
- Allows developers to control implementation details
  - Add details: 2D goes to 3D
  - Improve techniques: sequential sort goes to merge sort
- Users don’t worry about details
  - *There are programming constructs to support this*

All public everything

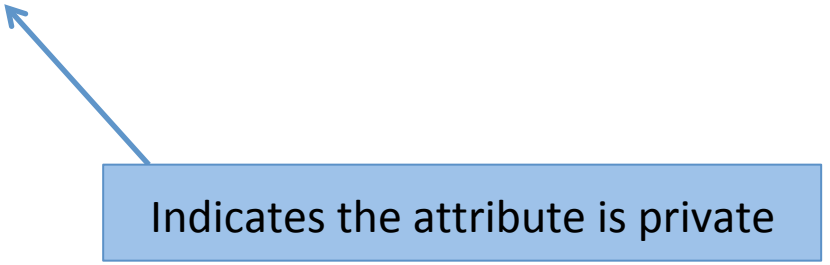
- Before, everything that was defined in the class was available to instances of that class
  - Both methods and attributes
- In particular attributes:
  - In all cases, they were “public” to instances

Privacy please

- As is the case in life, in programming sometimes we need to keep secrets
- Fortunately, this is supported *and encouraged* in OOP
  - private methods
  - private attributes
- Private object members are only “visible” from within the class
  - Only by `self`
  - Not even visible by subclasses!

The underscore

```
class Singer:
    __init__(self, name):
        self.name = name
        self.__last_night = 'wild'
```




```
bieb = Singer('Bieber')

print(bieb.name)

print(bieb.__last_night)
```

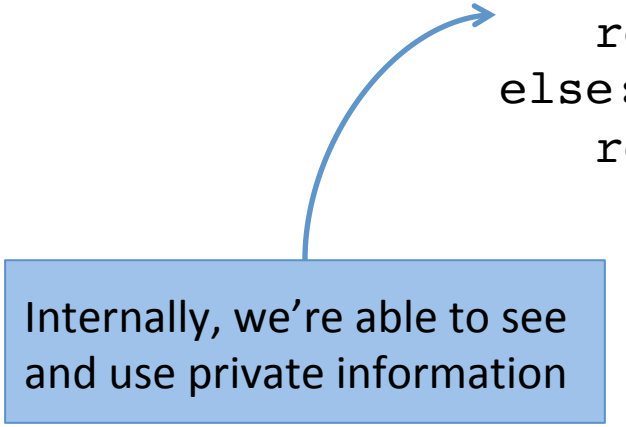
☺

☹

The underscore

```
class Singer:
    __init__(self, name):
        self.name = name
        self.__last_night = ‘wild’

    def about_last_night():
        if self.__last_night == ‘appropriate’:
            return self.__last_night
        else:
            return ‘nothing’
```



The underscore

```
class Singer:
    __init__(self, name):
        self.name = name
        self.__last_night = ‘wild’

    def __reveal_last_night(self):
        return False
```

Aside from the special (internal) methods, class authors can define their own privates methods as well

```
bieb = Singer(‘Bieber’)

print(bieb.__last_night)

bieb.__reveal_last_night()
```

☹

☹ ☹

Benefits of OOP

2. Make the job of a developer “easier”
   - Work with types rather than values
   - Build on others work

Specialization

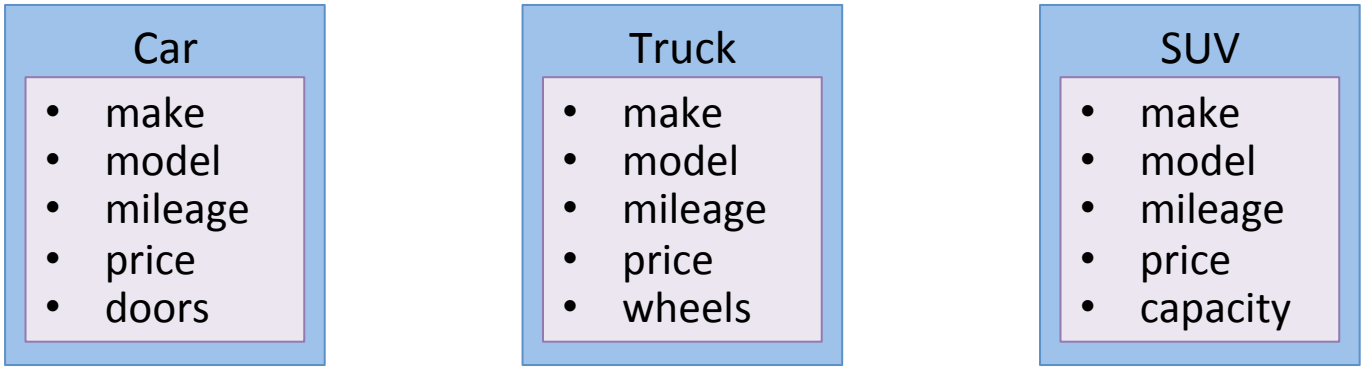
- One way to save work is to *specialize* things that have already been built:
  - An ordering of objects that are related, more increasingly specific

Example: cars

- You have to write a system that holds automobile information
- Specifically, you have to represent three types:
  - Cars: make, model, mileage, price, doors
  - Truck: make, model, mileage, price, wheels
  - SUV: make, model, mileage, price, capacity

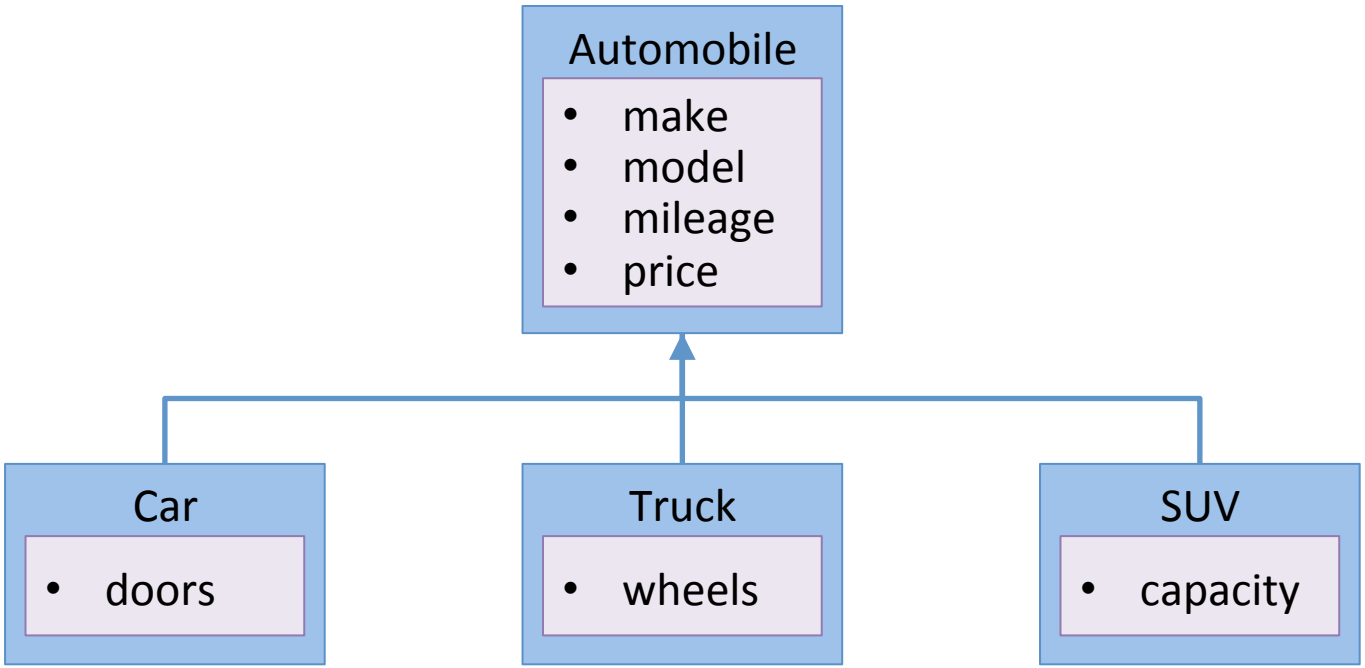
Solution 1

- Make an object for each
- Include five attributes for each

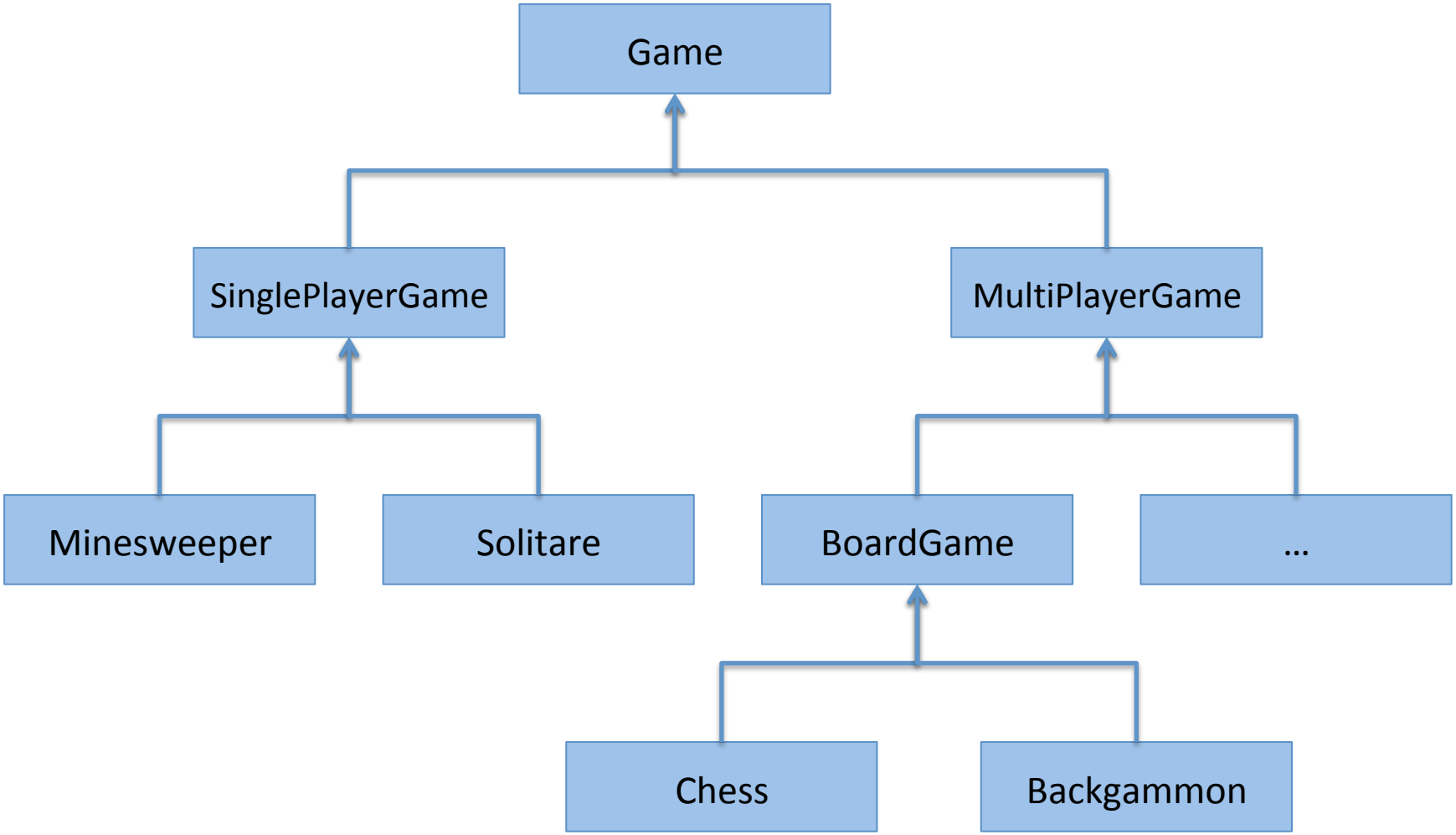


Solution 2

- Notice that they only differ by a one attribute
  - Build a parent class containing the common attributes
  - Have specialized classes for the differing attribute

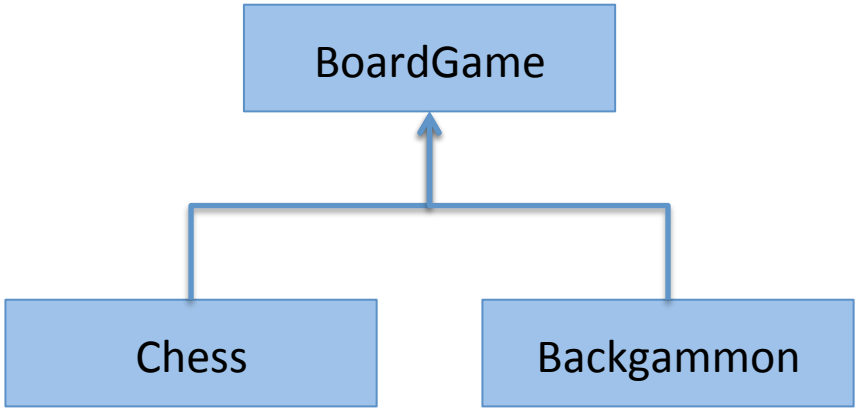


These can be elaborate

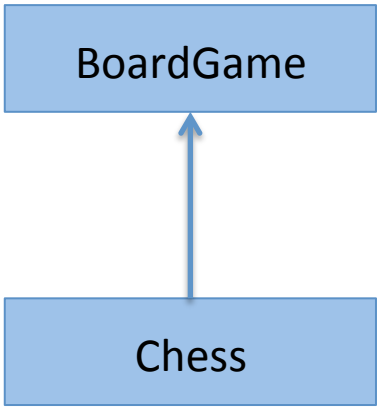


These can be elaborate

- "Elaborate" can mean complicated!
- No right way to do this
  - Depends on end goal
  - Experience
  - Perspective of the user
- Use the "is-a" test
  - Note the difference from "has-a"



Super and sub



A “parent” class is also known as a *super-class*

A “child” is also known as a *sub-class*

Super and sub

Commonly accepted that

- A sub-class is a specialization of a super-class
- A sub-class has all of the functionality of a super-class, and then some

Which makes you wonder

Instantiating a child

- If no constructor is present, the parents constructor will be called
- Otherwise, the child can explicitly call the parent constructor if it chooses

Benefits of OOP

3. Make the job of a developer “easier”
   - Work with types rather than values
   - Build on others work

The constructor example

- Recall our discussion of a child constructor
  - If present, use the child
  - Otherwise use the parent
- This rule applies to all methods!
- *Manifestation of specialization*

Overriding

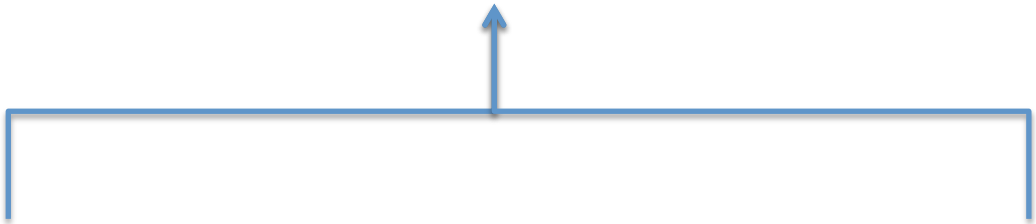
This allows us to do something cool:

- Recall that a sub-class contains all of the functionality of its respective super-class
- We can use this to make “generic” programs
  - Known as *polymorphism*: having more than one form
- Essentially: using a class through its parent interface — brilliant!

Example: animals

```
class Animal:
    def fur():
        return False

    def make_noise():
        return “Grrr”
```



```
class Dog(Animal):
    def fur():
        return True

    def make_noise():
        return “Woof”
```

```
class Bear(Animal):
    def fur():
        return self.maybe
```

Benefits of OOP

public/private

1. Encapsulate data and functionality into a single structure
   - Users don't worry about details
   - Developers are free to change their mind about details
2. Program with abstract concepts
   - Focus is on types

polymorphism

3. Make the job of a developer "easier"
   - Work with types rather than values
   - Build on others work

inheritance