# A class to represent employees.
class Employee:
    def get_hours(self):
        return 40
    def get_salary(self):
        return 40000.0
    def get_vacation_days(self):
        return 10
    def get_vacation_form(self):
        return "yellow"

# A class to represent secretaries.
class Secretary(Employee):
    def take_dictation(self, text):
        print("Taking dictation of text: " + text)

How many methods does Employee have?
How many attributes does Employee have?
What's the relationship between Secretary and Employee?
How many methods does Secretary have?
An __________ is an ____________ of a class.
**Terminology**

- Superclass
- Subclass

_____ is a subclass of _________

_____ is a superclass of _________

This is a Unified Modeling Language (UML) class diagram.
Employee regulations

• Consider the following employee regulations:
  • Employees work 40 hours / week.
  • Employees make $40,000 per year, except legal secretaries who make $5,000 extra per year ($45,000 total), and marketers who make $10,000 extra per year ($50,000 total).
  • Employees have 2 weeks of paid vacation leave per year, except lawyers who get an extra week (a total of 3).
  • Employees should use a yellow form to apply for leave, except for lawyers who use a pink form.

• Each type of employee has some unique behavior:
  • Lawyers know how to sue.
  • Marketers know how to advertise.
  • Secretaries know how to take dictation.
  • Legal secretaries know how to prepare legal documents.
Implementing Lawyer

• Consider the following lawyer regulations:
  • Lawyers get an extra week of paid vacation (a total of 3).
  • Lawyers use a pink form when applying for vacation leave.
  • Lawyers have some unique behavior: they know how to sue.

• Problem: We want lawyers to inherit *most* behavior from employee, but we want to replace parts with new behavior.
Overriding methods

- **override**: To write a new version of a method in a subclass that replaces the superclass's version.
  - No special syntax required to override a superclass method. Just write a new version of it in the subclass.

```python
class Lawyer(Employee):
    # overrides get_vacation_form method in Employee
    def get_vacation_form():
        return "pink"
    ...
```

- Exercise: Complete the **Lawyer** class.
  - (3 weeks vacation, pink vacation form, can sue)
Lawyer class

# A class to represent lawyers.
class Lawyer(Employee):
    # overrides get_vacation_form from Employee class
def get_vacation_form(self):
    return "pink"

    # overrides get_vacation_days from Employee class
def get_vacation_days(self):
        return 15  # 3 weeks vacation

def sue(self):
    print("I'll see you in court!")
Exercise: implement `Marketer`

- Recall the following marketer regulations:
  - Marketers make $10,000 more ($50,000 per year)
  - Marketers know how to market. (Print a phrase a marketer might use.)

- Write the code for the `Marketer class`
# A class to represent marketers.
class Marketer(Employee):
    def advertise(self):
        print("Act now while supplies last!")

    def get_salary(self):
        return 50000.0  # $50,000.00 / year
Levels of inheritance

• Multiple levels of inheritance are allowed.
  • Example: A legal secretary is the same as a regular secretary but makes more money ($45,000) and can file legal briefs
• Exercise: Complete the `LegalSecretary` class.
LegalSecretary class

# A class to represent legal secretaries.
class LegalSecretary(Secretary):
    def file_legal_briefs(self):
        print("I could file all day!")

    def get_salary(self):
        return 45000.0  # $45,000.00 / year
Change of perspective

• Recall the regulations regarding salaries:
  • Employees make $40,000 per year, except legal secretaries who make $5,000 extra per year ($45,000 total), and marketers who make $10,000 extra per year ($50,000 total).

• We've been hardcoding the salaries in the methods like this:
  ```python
def get_salary(self):
    return 45000.0  # $45,000.00 / year
```

• Instead, consider writing the methods in terms of a base salary plus an "uplift" :
  ```python
class LegalSecretary(Secretary):
    def get_salary(self):
      base_salary = ...regular employee salary...
      return base_salary + 5000.0
      ...
```
Calling overridden methods

• Subclasses can call overridden methods with `super`

\[
\text{super(ClassName, self).\text{method}(parameters)}
\]

• Example:

```python
class LegalSecretary(Secretary):
    def get_salary(self):
        base_salary = \text{super(LegalSecretary,self).get_salary()}
        return base_salary + 5000.0
    ...
```
Inheritance and constructors

• Imagine that we want to give employees more vacation days the longer they've been with the company.
  • For each year worked, we'll award 2 additional vacation days.

  • When an Employee object is constructed, we'll pass in the number of years the person has been with the company.

  • This will require us to modify our Employee class and add some new state and behavior.

• Exercise: Make necessary modifications to the Employee class.
Modified Employee class

class Employee:
    def __init__(self, initial_years):
        self.__years = initial_years

    def get_hours(self):
        return 40

    def get_salary(self):
        return 50000.0

    def get_vacation_days(self):
        return 10 + 2 * self.__years

    def get_vacation_form(self):
        return "yellow"
Problem with constructors

• Now that we've added the constructor to the Employee class, an error is produced:

```
TypeError: __init__() missing 1 required positional argument: 'initial_years'
```

• Short explanation: Once we write an `__init__(self, p1, … pn)` that requires parameters in the superclass, we must now write initialization methods for our employee subclasses as well.

• Exception: If the default behavior of the superclass is acceptable for all subclasses, you simply modify the class construction expression.
Modified Marketer class

# A class to represent marketers.
class Marketer(Employee):
    def __init__(years):
        super(Marketer, self).__init__(years)

    def advertise(self):
        print("Act now while supplies last!")

    def get_salary():
        return super(Marketer, self).get_salary() + 10000.0

• Exercise: Modify the Secretary subclass.
  • Secretaries' years of employment are not tracked.
  • They do not earn extra vacation for years worked.
Modified Secretary class

# A class to represent secretaries.
class Secretary(Employee):
    def __init__(self):
        super(Secretary, self).__init__(0)

    def take_dictation(self, text):
        print("Taking dictation of text: " + text)

• Since Secretary doesn't require any parameters to its constructor, LegalSecretary does not require a constructor.
  • Its default constructor calls the Secretary constructor.
Inheritance and attributes

• Try to give lawyers $5000 for each year at the company:

```python
class Lawyer(Employee):
    ...
    def get_salary(self):
        return super(Lawyer, self).get_salary() + 5000 * self.__years
    ...
```

• Does not work; the error is the following:

```
AttributeError: 'Lawyer' object has no attribute '_Lawyer__years'
```

• Private attributes cannot be directly accessed from subclasses.
  • One reason: So that subclassing can't break encapsulation.
  • How can we get around this limitation?
Improved Employee code
Add an accessor for any attribute needed by the subclass.

class Employee:
    self.__years

    def __init__(self, initial_years):
        self.__years = initial_years

    def get_years(self):
        return self.__years

...

class Lawyer(Employee):

    def __init__(self, years):
        super(Lawyer, self).__init__(years)

    def get_salary(self):
        return super(Lawyer, self).get_salary() + 5000 * get_years()
Revisiting Secretary

• The Secretary class currently has a poor solution.
  • We set all Secretaries to 0 years because they do not get a vacation bonus for their service.
  • If we call get_years on a Secretary object, we'll always get 0.
  • This isn't a good solution; what if we wanted to give some other reward to all employees based on years of service?

• Redesign our Employee class to allow for a better solution.
Improved Employee code

• Let's separate the standard 10 vacation days from those that are awarded based on seniority.

```python
class Employee:
    def __init__(self, initial_years):
        self.__years = initial_years

    def get_vacation_days(self):
        return 10 + self.get_seniority_bonus()

    # vacation days given for each year in the company
    def get_seniority_bonus(self):
        return 2 * self.__years
```

• How does this help us improve the Secretary?
Improved Secretary code

• Secretary can selectively override `get_seniority_bonus`; when `get_vacation_days` runs, it will use the new version.
  • Choosing a method at runtime is called *dynamic binding*.

```python
class Secretary(Employee):
    def __init__(self, years):
        super(Secretary, self).__init__(years)

    # Secretaries don't get a bonus for their years of service.
    def get_seniority_bonus(self):
        return 0

    def take_dictation(self, text):
        print("Taking dictation of text: " + text)
```