Refactoring Review

Name the refactorings

#1

BEFORE

```
def normalize(text):
  """Reformat some text"""
  text = text.trim()
  text = text.replace('_',
                      ' ')
  return text
```

AFTER

```
def normalize(text):
  """Reformat some text"""
  result = text.trim()
  result =
    result.replace('_',' ')
  return result
```

#2. (two names for this refactoring)

BEFORE

```
def roots(a, b, c):
  """Roots of Quadratic"""
  if b*b - 4*a*c >= 0:
    x1 = (-b +
    sqrt(b*b-4*a*c))/(2*a))
    x2 = (-b -
    sqrt(b*b-4*a*c))/(2*a))
    return (x1, x2)

  return None
```

AFTER

```
def roots(a, b, c):
  """Roots of Quadratic"""
  descrim = b*b - 4*a*c
  if descrim >= 0:
    descrim = sqrt(descrim)
    x1 = (-b + descrim)/(2*a)
    x2 = (-b - descrim)/(2*a)
    return (x1, x2)

  return None
```

#3

BEFORE

```
def find(text: str):
  """Find text in file"""
  found = False
  line = None
  file = open("somefile")
  while not found:
     line = file.readline()
     if text in line:
        found = True
  file.close()
  return line
```

AFTER

```
def find(text: str):
  """Find text in file"""
  with open("somefile")
       as file:
     for line in file:
        if text in line:
           return line

  return None
```

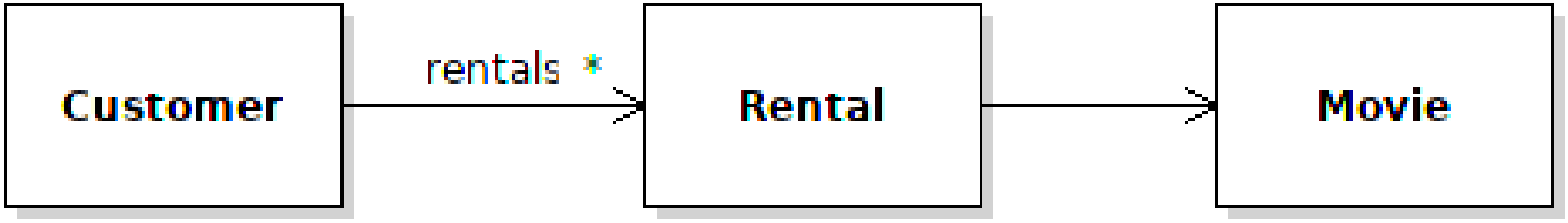
#4

BEFORE

```
title = rental.get_movie()\
                 .get_title()
```

AFTER

```
title = rental.get_title()
```



#5

BEFORE

```
person[0] = 'Bill'
person[1] = 'Gates'
person[2] = 'bill@msft.com'


print_person(person)

def print_person(person):
   print(f"{person[0]}
           {person[1]}
     email <{person[2]}>")
```

AFTER

```
class Person:
  def __init__(self,
     first, last, email):
      self.first = first
person = Person("Bill",...)
print_person(person)

def print_person(person):
   print(f"{person.first}
           {person.last}
   email <{person.email}>")
```

#6

BEFORE

```
def print_person(firstname,
                 lastname,
                 email):
   print(f"{firstname}
           {lastname}
           email <email>")


# invoke using:
p = Person("Bill","Gates"..
print_person(p.firstname,
   p.lastname, p.email)
```

AFTER

```
def print_person(person):

   print(f"{person.first}
          {person.last}
   email <{person.email}>")


# invoke using:
p = Person("Bill","Gates"..
print_person( p )
```

#7

BEFORE

```
class Person:
  def __init__(self,
     first, last, email):
      self.first = first

def print_person(person):
   print(f"{person.first}
           {person.last}
   email <{person.email}>")

person = Person("Bill",...)
print_person(person)
```

AFTER

```
class Person:
  def __init__(self,
     first, last, email):
      self.first = first

  def __str__(self):
     return f"{self.first}
     {self.last} email ..."

person = Person("Bill",...)
print(person)
```

what is the *justification* (reason) for this change?

#8

BEFORE

```
def greet(firstname):
   if datetime.now().hour
        < 12:
     print("Good morning",
           firstname)
   else:
     print("G'd afternoon",
           firstname)
```

AFTER

```
def greet(firstname):
   if is_morning():
     print("Good morning",
           firstname)
   else:
     print("G'd afternoon",
           firstname)

def is_morning():
  return \
    datetime.now().hour < 12
```

#9

BEFORE

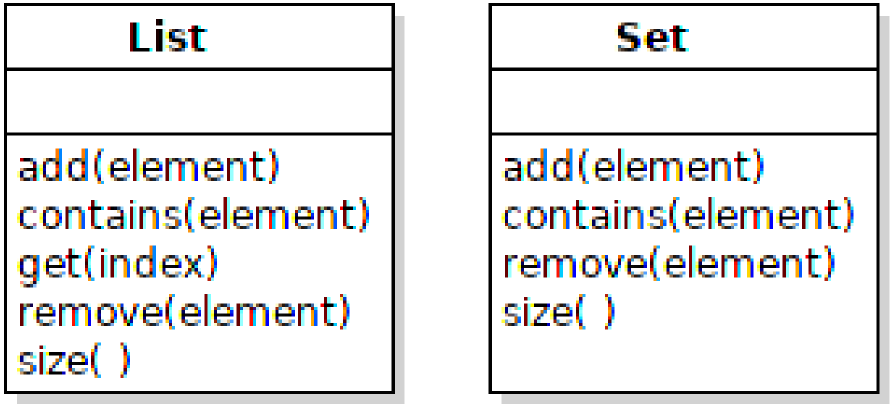
```
game = Game(800, 600)
```

AFTER

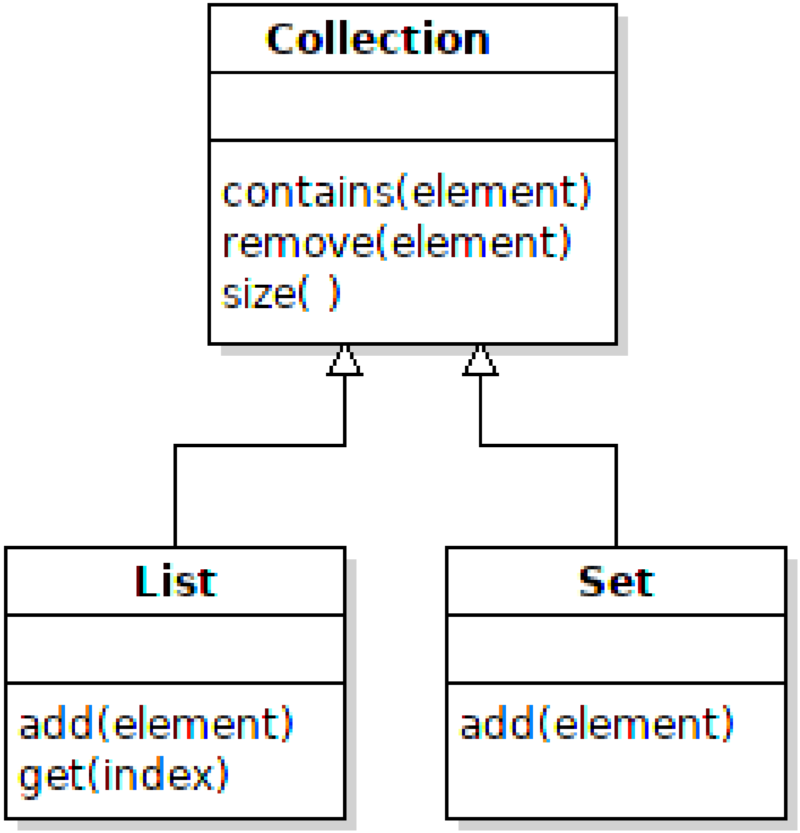
```
CANVAS_WIDTH = 800
CANVAS_HEIGHT = 600

game = Game(CANVAS_WIDTH,
            CANVAS_HEIGHT)
```

#10

BEFORE



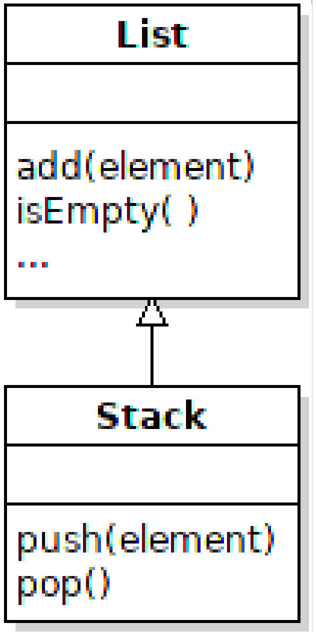
AFTER



Why not move `add(element)` to Collection, too?

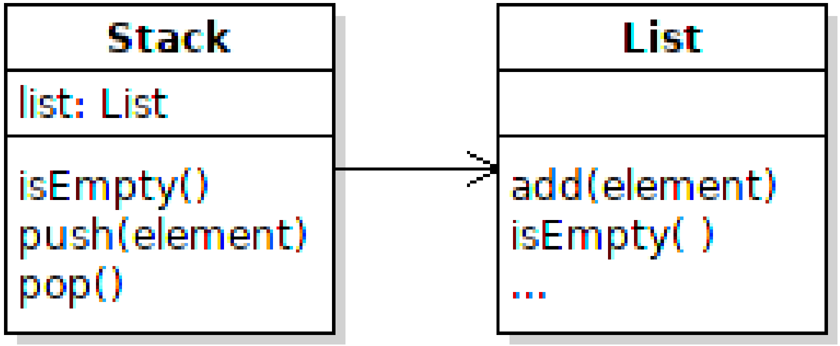
#11

BEFORE



```
class Stack(List):

  def push(self, e):
      super().add(e)
```

AFTER



```
class Stack:

  def push(self, e):
      self.list.add(e)
```

After: `Stack` must implement `isEmpty()`, too.

Why Not Stack extends List?

O-O Basics:

- A Stack *is not* a List. Fails the "*is a*" test.

Design Principles:

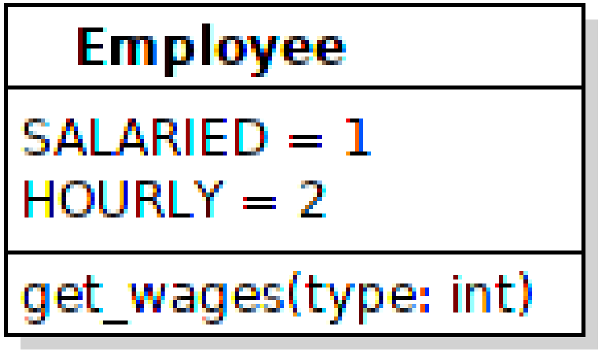
- Prefer *Composition over Inheritance*, also called
- *Prefer Delegation over Inheritance*

Code Symptom:

- *Refused Bequest* - Stack doesn't use most List methods

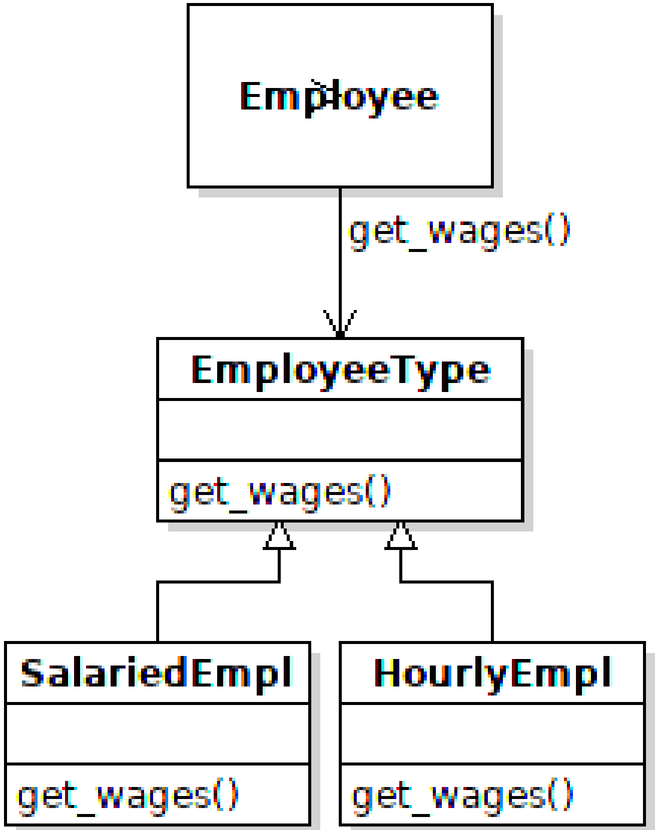
#12 (two names for this refactoring)

BEFORE



```
def get_wages(self, type):
   if type == SALARIED:
      ...
   elif type == HOURLY:
      ...
```

AFTER



#13

BEFORE

```
bird = 0
cat = 1
dog = 2
def speak(species):
   if species == bird:
     print("chirp, chirp")
   elif species == cat:
     print("meow")
   elif species == dog:
     print("woof, woof")
   else: ...
```

AFTER

```
species = Zoo.get("cat")

species.speak()

class Cat(Animal):
   def speak(self):
      print("meow")

class Dog(Animal):
   def speak(self):
      print("woof, woof")
```

#14

BEFORE

```
SPADES = 1
HEARTS = 2
CLUBS  = 3
DIAMONDS = 4

class Card:
  def __init__(self, value,
              suite: int):
      ...

c = Card(4, HEARTS)
```

AFTER

```
class Suite(Enum):
  SPADES = 1
  HEARTS = 2
  CLUBS  = 3
  DIAMONDS = 4

class Card:
  def __init__(self, value,
            suite: Suite):
      ...
c = Card(4, Suite.HEARTS)
```

This refactoring is different from #11 and #12.

Why Refactor?

For each refactoring, state the benefit(s) of it.

Be specific.

Avoid vague claims like "*easier to ...*". Instead, state why and how something is "*easier*".

Extract Method

Benefits:

- increase opportunity to reuse code and eliminate duplicate code
- make method easier to understand, which reduces errors and improves maintainability
- by reducing the amount of work a method is doing, it gets closer to the goal of "1 method does only 1 thing", and make make for more descriptive method name