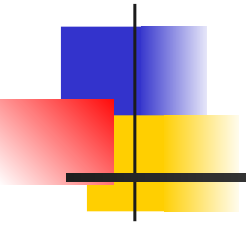


Refactoring Review

Instructions

Write the name of refactoring examples.
Write the name *exactly* as shown on `refactoring.guru`.

https://refactoring.guru/refactoring/techniques

If there are *two possible answers, then write the name of either* ***<u>one</u>*** *refactoring.*

The *Refactoring Category* is shown
at the bottom of each slide.

#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
```

Refactoring Category: Composing Methods

#2 (two possible answers)

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
```

Simplifying Conditional Expressions

(many students write code like on the left)

#4

BEFORE

```
# chain calls to get title


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

AFTER

```
# Rental gets title from
# movie and returns it.
# Movie still has get_title
title = rental.get_title()
```

Moving Features Between Objects

#5

BEFORE

```
first = 'Bill'
last  = 'Gates'
email = 'bill@msft.com'

print_person(
    first, last, email)

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

AFTER

```
@dataclass
class Person:
    first: str
    last: str
    email: str
p = Person("Bill","Gates",...)
print_person(p)

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

Simplifying Method Calls

#6

BEFORE

```
def print_rental(title,
        days_rented, price):
    print("{:20s} {:6d} {:f}"
      .format(title,
              days_rented,
              price))


# Usage:
r = Rental("Frozen", 3)
print_rental(r.get_title(),
    r.get_days_rented(),
    r.get_price())
```

AFTER

```
def print_rental(r: Rental):
    print("{:20s} {:6d} {:f}"
      .format(
          r.get_title(),
          r.get_days_rented(),
          r.get_price()))


# Usage:
r = Rental("Frozen", 3)
print_rental(r)
```

Simplifying Method Calls

#7

BEFORE

```
def vote(question, choice):
  if not question.can_vote():
    messages.error(
        "voting not allowed")
  elif choice not in
      question.choice_set():
      messages.error("invalid ...")
  else:
      Vote.objects.create(
           user=user, question=...)
      return redirect('polls:result')
  # if any error, redirect to
detail
  return
redirect('polls:detail',...
```

AFTER

```
def vote(question, choice):
  if not question.can_vote():
     messages.error(
        "voting not allowed")
     return redirect('polls:detail',..
  if choice not in \
        question.choice_set():
     messages.error("invalid ...")
     return redirect('polls:detail',..
  Vote.objects.create(
       user=user, question=...)
  return redirect('polls:result',...)
```

Simplifying Conditional Expressions

#8 (two possible answers)

BEFORE

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

AFTER

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

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

1. Simplifying Conditional Expressions
2. Composing Methods

#9

BEFORE

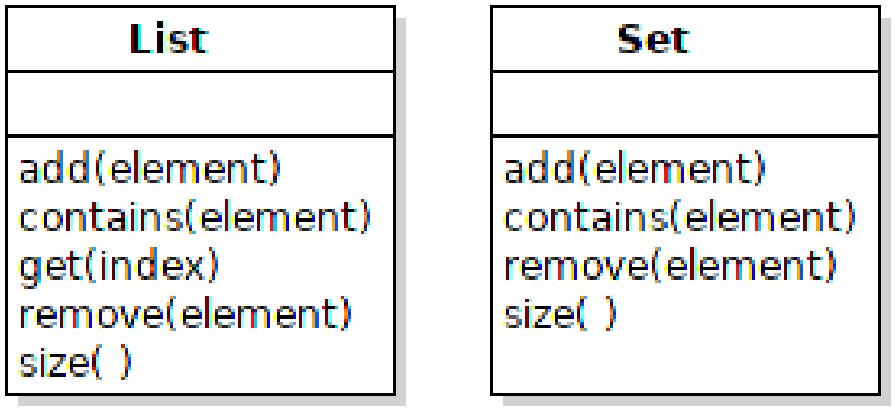
```
game = Game(800, 600)
```

AFTER

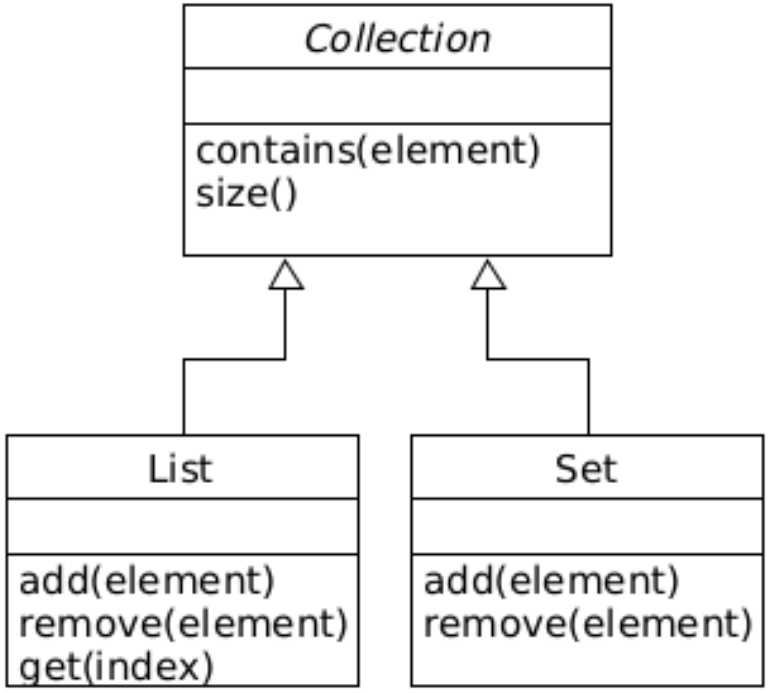
```
CANVAS_WIDTH = 800
CANVAS_HEIGHT = 600

game = Game(CANVAS_WIDTH,
            CANVAS_HEIGHT)
```

#10

BEFORE



AFTER

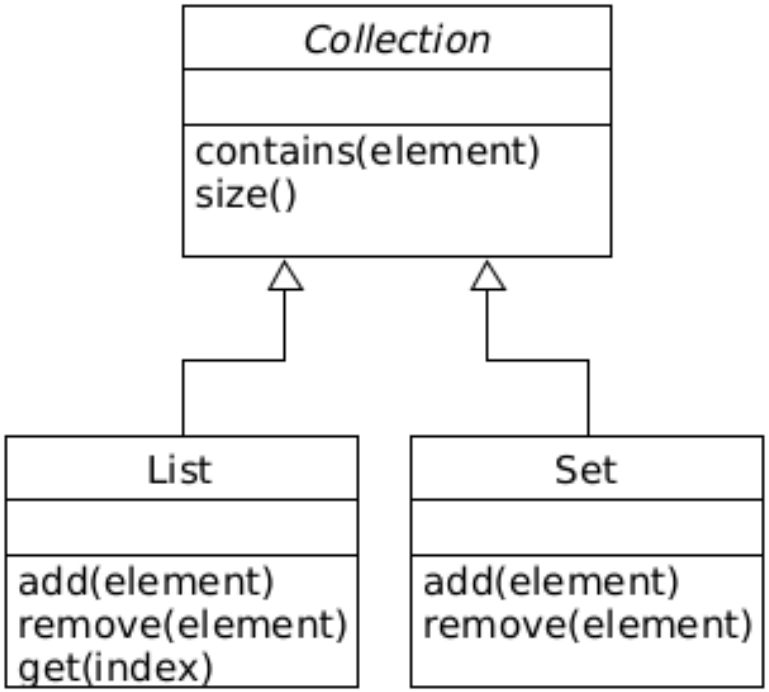


Dealing with Generalization

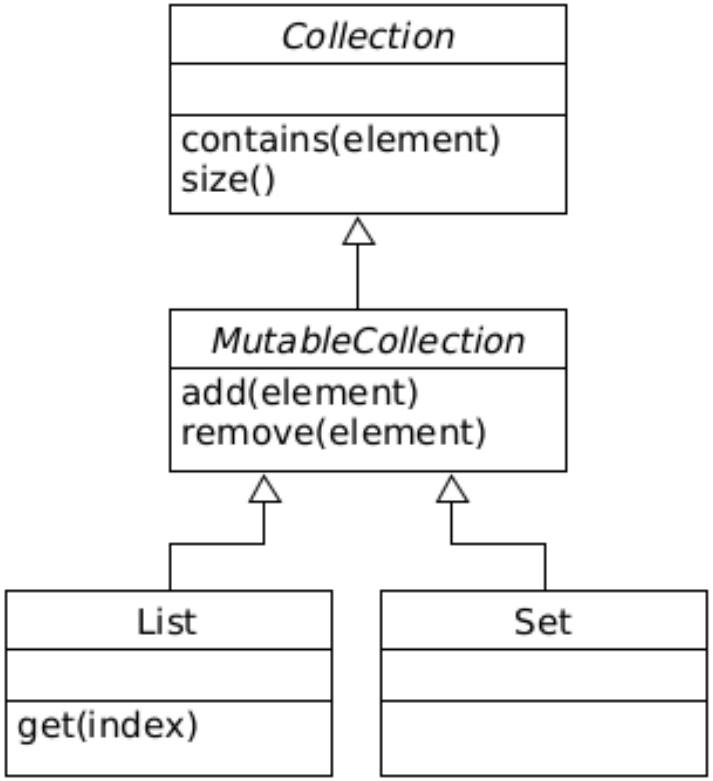
#11

BEFORE

Same code in many collections.



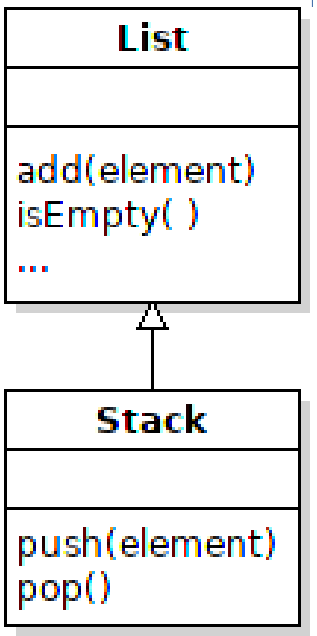
AFTER



Dealing with Generalization

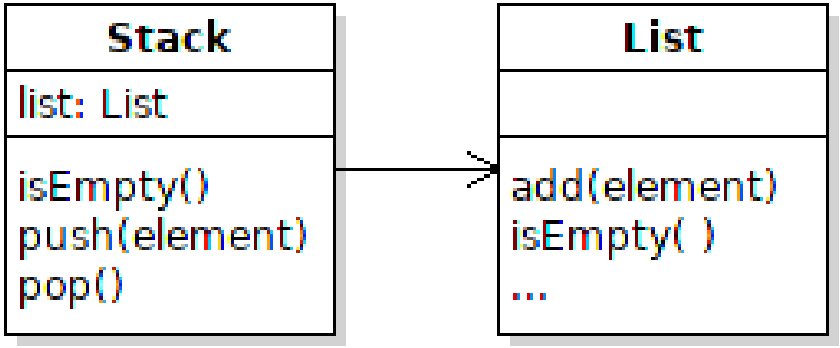
#12

BEFORE




```
class Stack(List):
  def push(self, e):
      super().append(e)
```

AFTER




```
class Stack:
  def push(self, e):
      self.list.append(e)
```

Dealing with Generalization

#13 Name Two Refactorings

BEFORE

```
class Rental:
  def get_price(self):
     if type == NEW_RELEASE:
        price = 3*self.days
     elif type == CHILDREN:
        price = 1.5 + \
        1.5*max(0, self.days-3)
     else:
        price = ...
     return price
```

AFTER

```
class Rental:
     days: int
     price_code: PriceCode

     def get_price(self):
       return self.price_code.\
          get_price(self.days)

class PriceCode(ABC):
     pass
class NewRelease(PriceCode):
     def get_price(self, days):
          return 3*days
```

1. Organizing Data, 2. Simplifying Conditional Expressions

#13 Hint

Answer is not *Replace Type Code with Subclass*

There are also classes (not shown to save space)

```
class ChildrensMovie(PriceCode):
    def get_price(self, days): ...

class RegularMovie(PriceCode):
    def get_price(self, days): ...
```

#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)
```

*Organizing Data, but **different refactoring from #13.***

#15

BEFORE

```
@dataclass
class Person:
    name: str
    telephone: str
```

AFTER

```
@dataclass
class Person:
    name: str
    telephone: Telephone
@dataclass
class Telephone:
    country_code: CountryCode
    phone_number: digits
    extension:    str
class CountryCode(enum.Enum):
    Thailand = 66
```

*Organizing Data, but **different refactoring from #13-14.***

Can You Justify Your Refactorings?

Imagine refactoring during a code review.

Can you explain to the team ***why you refactor****?*

You *should* be able to:

- Explain the Benefit of each refactoring
- Be specific in your reason - no vague claim like "*easier to ...*"

  Instead, state why and how something is "*easier*".

Example: Extract Method

Why? What's the Benefit?

- method *logic* becomes clearer, which reduces errors and improves maintainability
- the code you extract can be **tested separately**. When it is embedded in another method, it might not be testable.
- by reducing the amount of work a method is doing, it gets closer to the goal of "1 method does 1 thing". And the method name can be **more descriptive**.
- increase opportunity to **reuse code** and eliminate duplicate code - focused methods are more reusable

Refactoring is Not Always this Simple

These examples are simple
in order to fit on one slide.

Actual code is much more complex.

...and the more complex the code is,
the more it may need refactoring.

It helps to know

1) refactoring signs and symptoms,

2) design principles.