Concepts of Inheritance

What is Inheritance?

An act of receiving something passed down from your parents to you.

- Legacy
- Hereditament
- so on

Inheritance in Computer Science

Vangie Beal

said

- In object-oriented programing (OOP) inheritance is a feature that represents the **"is a"** relationship between different classes.
- Inheritance allows a class to have the same behavior as another class and extend that behavior to provide special action for specific needs.

Credits - Image from internet

Inheritance Flow Diagram

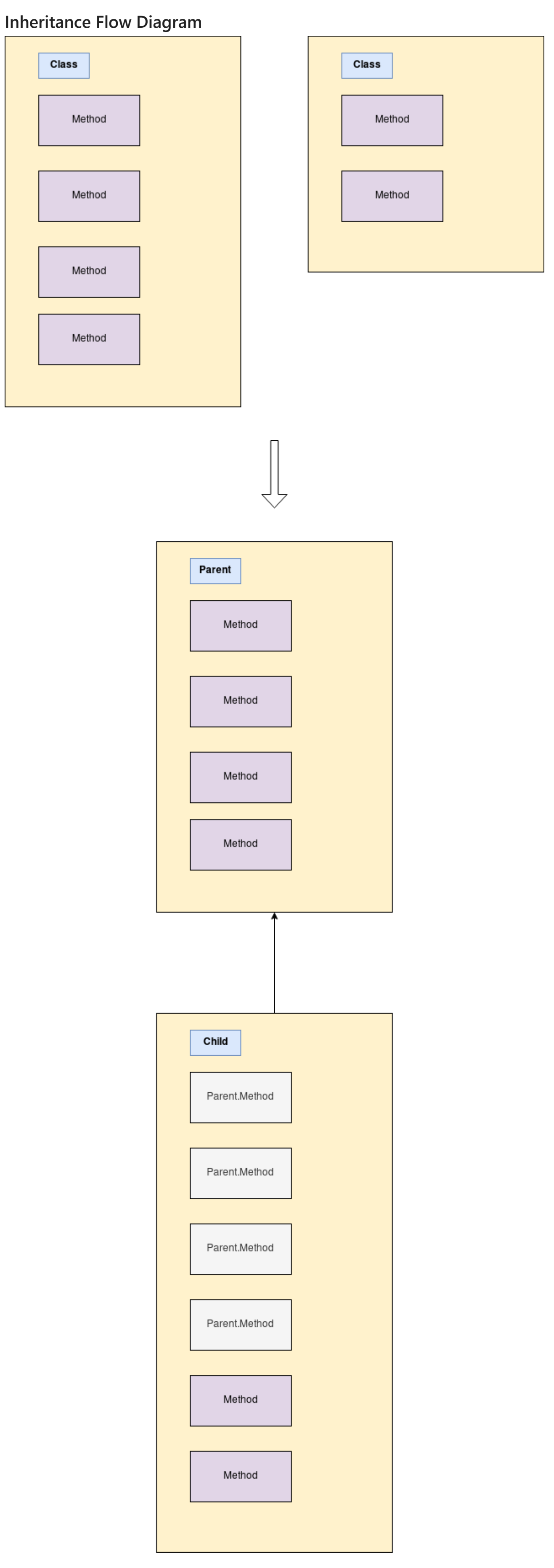

Image by author

Types of Inheritance

Inheritance is one of the important concepts in implementing Object Oriented Programming.

- Single-level Inheritance
- Multi-level Inheritance
- Multiple Inheritance

Rules

- There should be same attributes of `__init__()` methods in both the `Parent` class and `Child` class.
- Once a `Child` class inherits the `Parent` class,
  - all the methods of `Parent` class can be called using `self.<method_name()>` technique.
  - object creation should be done using `Child` class not the `Parent` class.

Note - Same rules apply to **Multi-level** and **Multiple** inheritance.

Single-level Inheritance

- If only one **base class** is used to derive only one **subclass**, it is known as **single level** inheritance.

Structure

```
class A():
    def __init__(self):
        # do something
        pass

class B(A):
    def __init__(self):
        # do something
        pass
```

In [1]:

```
class College():
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def status_report(self):
        return "The time starts from {} till {}.".format(self.start, self.end)

###########################

class MathFest(College):
    def __init__(self, start, end):
        self.start = start
        self.end = end

    def math_status_report(self):
        open_statement = "The Status report for Math fest is as follows."
        print(open_statement)
        statement = self.status_report()
        return statement
```

In [2]:

```
math = MathFest(start="12 PM", end="6 PM")
print(math.math_status_report())
print(math.status_report())
```

```
The Status report for Math fest is as follows.
The time starts from 12 PM till 6 PM.
The time starts from 12 PM till 6 PM.
```

Multi-level Inheritance

- **Multilevel inheritance** refers to a mechanism where one class can inherit from a **derived class**, thereby making this **derived class** the **base** class for the new class.

Structure

```
class A():
    def __init__(self):
        pass

class B(A):
    def __init__(self):
        pass

class C(B):
    def __init__(self):
        pass
```

In [3]:

```
# from time import sleep
# from random import choice

# a = [1, 2 ,4, 5, 6, 7, 8, 9, 13, 12, 15]
# for i in a:
#     print(choice(a))
#     sleep(1)
```

In [4]:

```
from time import sleep
from random import choice

time_range = list(range(2, 8)) # [2, 3, 4, 5, 6, 7]

## Grandparent class
class WashingMachine():
    def __init__(self, clothes):
        self.clothes = clothes

    def begin_stage(self):
        machine_desc = "This machine has python os 😉 which detects clothes to wash."
        print(machine_desc)
        print("Clothes provided to the machine - {}".format(self.clothes))
        sleep(3)
        return "{} is yet to be washed.".format(self.clothes)


## Parent class
class SpinningMachine(WashingMachine):
    def __init__(self, clothes):
        self.clothes = clothes

    def spinning_stage(self):
        print(self.begin_stage())
        print('\n--------\n')
        motor_desc = "The battery of this machine has python components 😋 that help spin the clothes."
        print(motor_desc)
        time_requires = choice(time_range)
        print("\t Total time required : {} secs".format(time_requires))
        sleep(time_requires)
        return "{} is getting spun to clean the dirt.".format(self.clothes)


## Child class or Grandchild class
class DryingMachine(SpinningMachine):
    def __init__(self, clothes):
        self.clothes = clothes

    def drying_stage(self):
        print(self.spinning_stage())
        print('\n--------\n')
        dry_desc = "This machine has mini-sun of python version to dry all the clothes 😍."
        print(dry_desc)
        time_requires = choice(time_range)
        print("\t Total time required : {} secs".format(time_requires))
        sleep(time_requires)
        return "{} is getting dried.".format(self.clothes)
```

Connections for the above example

- DryingMachine → `is a` → SpinningMachine
- SpinningMachine → `is a` → WashingMachine
- ...
- DryingMachine → `is a` → SpinningMachine → `is a` → WashingMachine

In [5]:

```
my_clothes = ["T-shirt", "Jeans", "Towel"]
cloth = choice(my_clothes)

machine = DryingMachine(clothes=cloth)
print(machine.drying_stage())
print("\n\n{} washed 🥳 successfully.".format(cloth))
```

```
This machine has python os 😉 which detects clothes to wash.
Clothes provided to the machine - Jeans
Jeans is yet to be washed.

--------

The battery of this machine has python components 😋 that help spin the clothes.
         Total time required : 5 secs
Jeans is getting spun to clean the dirt.

--------

This machine has mini-sun of python version to dry all the clothes 😍.
         Total time required : 5 secs
Jeans is getting dried.


Jeans washed 🥳 successfully.
```

Multiple Inheritance

- **Multiple inheritance** is a unique feature in which an object or class can inherit characteristics and features from more than one parent object or parent class.

Structure

```
class A():
    def __init__(self):
        pass

class B():
    def __init__(self):
        pass

######

class C(A, B):
    def __init__(self):
        pass
```

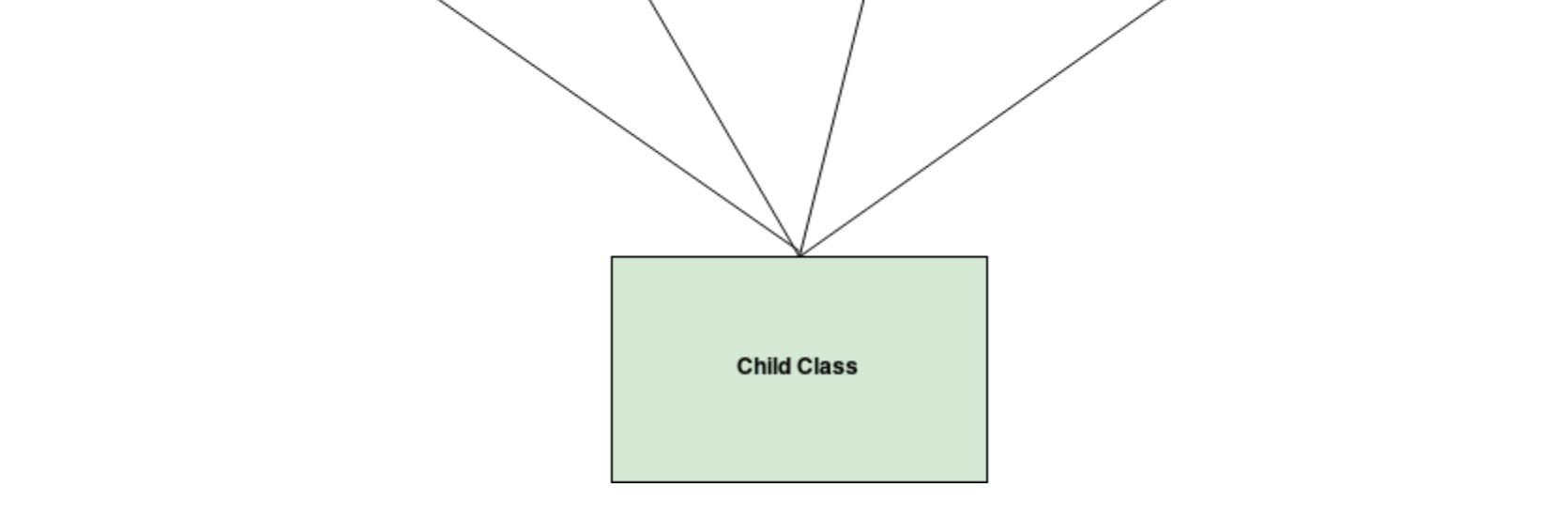

Image by author

In [6]:

```
class HumanBeings():
    def human_popn(self):
        return "There are about 7.8 billion people"

###########################

class Insects():
    def insect_popn(self):
        return "There are about 10 quintillion insects"

###########################

class Animals():
    def animal_popn(self):
        return "There are about 8.7 million species of animals"

###########################

class Birds():
    def bird_popn(self):
        return "There are about 200 to 400 billion individual birds"

###########################

class Trees():
    def tree_popn(self):
        return "There are about 3 trillion trees"

###########################

class Earth(Trees, Birds, Animals, Insects, HumanBeings):
    def planet_life(self):
        print(self.tree_popn() + ' living on earth.')
        print(self.bird_popn() + ' living on earth')
        print(self.insect_popn() + ' living on earth.')
        print(self.animal_popn() + ' living on earth.')
        print(self.human_popn() + ' living on earth.')
        print('###########################')
        return 'Earth is beautiful.'
```

In [7]:

```
life = Earth()
print(life.planet_life())
```

```
There are about 3 trillion trees living on earth.
There are about 200 to 400 billion individual birds living on earth
There are about 10 quintillion insects living on earth.
There are about 8.7 million species of animals living on earth.
There are about 7.8 billion people living on earth.
###########################
Earth is beautiful.
```

Other Concepts in OOPs

- Abstraction
- Encapsulation
- Polymorphism

Note - Please read about this by yourself and ask doubts in case there is a confusion.

In [8]:

```
class Myself():
    def __init__(self, name, interest):
        self.name = name
        self.__interest = interest

    # private method → __
    def __show_yourself(self):
        return "The name is {} and interest is {}".format(self.name, self.__interest)

# me = Myself(name="sameer", interest="python")
# print(me.__show_yourself())

# obj._ClassName-method_name
# print(me._Myself__show_yourself())
```

Polymorphism

- Polymorphism is the ability of a programming language to present the same **functionality** for several different underlying data types.
- The best example for this is `len()` function in Python.

In [9]:

```
## String
print("For string \t", len("hello"))                  # len() with str

## List
print("For list \t", len([1, 2, 12, 32, 43, 56]))              # len() with list

## Tuple
print("For tuple \t", len((1, 2, 0, 0, 0,)))                   # len() with tuple

## Dictionary
print("For dictionary \t", len({1 : 2, 3: 4, "hello" : "hi"})) # len() with dictionary
```

```
For string       5
For list         6
For tuple        5
For dictionary   3
```

In [10]:

```
def add_(e):
    if isinstance(e, int):
        return e + 3
    elif isinstance(e, float):
        return e + 3
    elif isinstance(e, str):
        return e + str(3)
```

In [11]:

```
add_(1)
```

Out[11]: 4

In [12]:

```
add_(1.2)
```

Out[12]: 4.2

In [13]:

```
add_("python ")
```

Out[13]: 'python 3'

What did we learn?

- Inheritance definition
- Inheritance flow diagram
- Types of inheritance with coding examples