Object-oriented programming

Until now, we have not been doing much with objects. This chapter discusses **objects** and **classes**, and how an object-oriented language like Java is different from languages that are not object-oriented.

Chapter Topics:

- What is an Object?
- What is a Class?
- Characteristics of Objects
- Static Methods
- Constructors
- Cookie Cutters
- Dot Notation

Forget programming for a while. Think about the World and the things that are in it. What things are objects? What things are not objects? This is actually a difficult problem which has occupied philosophers for thousands of years. Don't be too worried if it is not immediately clear to you.

Object-oriented programming

- Object-oriented programming is a design philosophy.
- Its abbreviation is OOP .
- In this programming stile, a program see the problem as the problem is seen in the real word .
- Object-oriented programming (OOP) is a programming language model organized around objects rather than "actions" and data rather than logic. Historically, a program has been viewed as a logical procedure that takes input data, processes it, and produces output data.

Object-oriented programming

- The concepts and rules used in object-oriented programming are based on the following features:
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

Object-oriented programming

- Abstraction
  - Data abstraction refers to, providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

- Encapsulation
  - Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.

Object-oriented programming

- Inheritance
  - One of the most useful aspects of object-oriented programming is code reusability. As the name suggests Inheritance is the process of forming a new class from an existing class that is from the existing class called as base class, new class is formed called as derived class.
- Polymorphism
  - The ability to use an operator or function in different ways in other words giving different meaning or functions to the operators or functions is called polymorphism. Poly refers to many. That is a single function or an operator functioning in many ways different upon the usage is called polymorphism.

Objects

- This is the basic unit of object oriented programming. That is both data and function that operate on data are bundled as a unit called as object.

List four objects and list four non-objects.

	Objects	Non-objects
1	A pen	The upper 37% of the pen
2	A computer keyboard	The air above the keyboard
3	A shoe	The color of the shoe
4	A desk	All desks in the world

What makes an object

It is easier to list things that are objects than to list things that are not objects. Just to talk about something seems to make it an object, somehow. René Descartes (the 17th century philosopher) observed that humans view the world in object-oriented terms. The human brain wants to think about objects, and our thoughts and memories are organized into objects and their relationships. Perhaps non-human brains work differently.

One of the ideas of object-oriented software is to organize software in a way that matches the thinking style of our object-oriented brains. Instead of machine instructions that change bit patterns in main storage, we want "things" that "do something." Of course, at the machine level nothing has changed—bit patterns are being manipulated by machine instructions. But we don't have to think that way.

Characteristics of objects

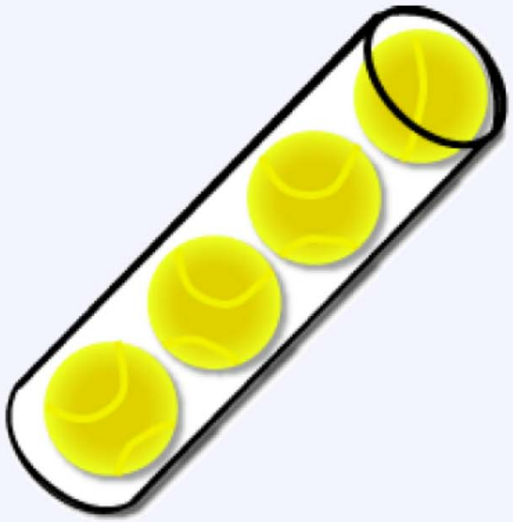
The first item in this list is too restrictive. For example, you can think of your bank account as an object, but it is not made of material. (Although you and the bank may use paper and other material in keeping track of your account, your account exists independently of this material.) Although it is not material your account has properties (a balance, an interest rate, an owner) and you can do things to it (deposit money, cancel it) and it can do things (charge for transcactions, appreciate interest).

The last three items on the list seem clear enough. In fact, they have names:

- An object has **identity** (it acts as a single whole).
- An object has **state** (it has various properties, which might change).
- An object has **behavior** (it can do things and can have things done to it).

This is a somewhat ordinary description of what an object is like. (This list comes from the book *Object-oriented Analysis and Design*, by Grady Booch, Addison-Wesley, 1994.) Do not be surprised if other notes and books have a different list. When you start writing object-oriented software you will find that this list will help you decide what your objects should be.

Consider a tube of four yellow tennis balls.



- Is the tube of tennis balls an object?
  Yes. It has identity (my tube of balls is different than yours), it has state (opened, unopened, brand name, location), and behavior (although not much).
- Is each tennis ball an object?
  Yes. It is OK for objects to be part of other objects. Although each ball has nearly the same state and behavior as the others, each has its own identity.
- Could the top two balls be considered a single object?
  Not ordinarily. Each has its own identity independent of the other. If they were joined together with a stick you might consider them as one object.
- Is the color of the balls an object?
  No. It is a property of each ball.
- Is your understanding of tennis balls an object?
  Probably not, although it is unclear what it is. Perhaps it is a property of the object called "your brain."

Software objects

Many programs are written to do things that are concerned with the real world. It is convenient to have "software objects" that are similar to "real world objects". This makes the program and what it does easier to think about. Software objects have *identity*, *state*, and *behavior* just as do real world objects. Of course, software objects exist entirely within a computer system and don't directly interact with real world objects.

Software objects as memory

Objects (real world and software) have *identity*, *state*, and *behavior*.

Software objects have **identity**. Each is a distinct chunk of memory. (Just like a yellow tennis ball, each software object is a distinct individual even though it may look nearly the same as other objects.)

Software objects have **state.** Some of the memory that makes up a software object is used for variables which contain values. These values are the state of the object.

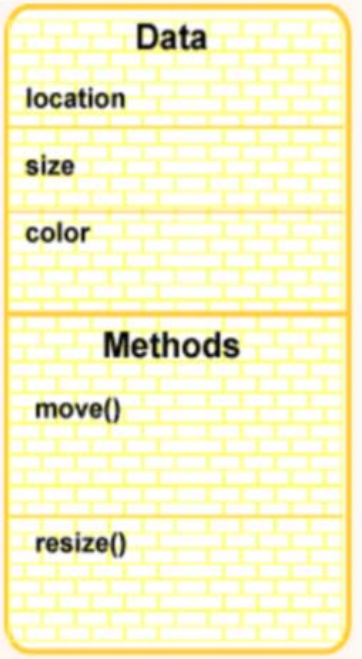
Software objects have **behavior.** Some of the memory that makes up a software object contains programs (called *methods*) that enable the object to "do things". The object does something when one of its method runs.

Picture of an object

In terms of object-oriented programming, a **von Neumann computer** uses general purpose memory to store both the state and behavior of objects. It is interesting that an idea from the 1940's is still important.

A software object consists of both variables (state information) and methods (recipies for behavior). In the picture, the yellow bricks represent bytes of memory out of which the object is built. This object has some variables, `location`, `color`, and `size`, and has some methods that control its behavior.

In object-oriented programming, the programer uses a programming language (such as `Python`) to describe various objects. When the program is run (after being compiled) the objects are created (out of main storage) and they start "doing things" by running their methods.



Class

- Classes are designed to create and manage new data types and objects.
- Syntax:

  class name:
- Each class is an object
- We also create new object from a class
- This new object called an instance
- Each class has some attribute: variables and methods
- Attributes are access by dot notation

  name.att

Class

```
class rec:
    x=3
    y=5

print(rec.x)
print(rec.y)

rec.x=12
rec.y='test'
print(rec.x)
print(rec.y)

rec.y = rec.y + ' a class'
print(rec.y)
```

```
3
5
12
test
test a class
```

Class

```
class rec: pass

rec.x=12
rec.y='test'
print(rec.x)
print(rec.y)
rec.y = rec.y + 'a class'
print(rec.y)
```

```
12
test
test a class
```

Class

```
class rec:
   x=3
   y=5

print('rec.x: ', rec.x)
print('rec.y: ', rec.y)

rec.w=12
rec.z='test'

print('rec.w: ', rec.w)
print('rec.z: ', rec.z)

print('rec.x: ', rec.x)
print('rec.y: ', rec.y)
```

```
rec.x:  3
rec.y:  5
rec.w:  12
rec.z:  test
rec.x:  3
rec.y:  5
```

Class general form

```
class name(superclass,...):          # Assign to name
    attr = value                     # Shared class data
    def method(self,...):            # Methods
        self.attr = value            # Per-instance data
```

Class general form

```
class FirstClass:                    # Define a class object
        x=5
        def setdata(self, value):    # Define class's methods
            self.x = value           # self is the instance
        def display(self):
            print(self.x)            # self.data: per instance


print(FirstClass.x)
FirstClass.display(FirstClass)

FirstClass.setdata(FirstClass,15)
FirstClass.display(FirstClass)

#ouyput

   5
   5
   15
```

Class general form

```
class FirstClass:
        x=5
        def setdata(self, value):
            self.x = value
        def display(self):
            print(self.x)


print(FirstClass.x)
FirstClass.display(FirstClass)

FirstClass.setdata(FirstClass,15)
FirstClass.display(FirstClass)

#ouyput

   5
   5
   15
```

Class general form

```
class FirstClass:                    # Define a class object
        x=5
        def setdata(self, value):    # Define class's methods
            self.x = value           # self is the instance
        def display(self):
            print(self.x)            # self.data: per instance


print(FirstClass.x)
FirstClass.display(FirstClass)

FirstClass.setdata(FirstClass,15)
FirstClass.display(FirstClass)

FirstClass.y='test'
print(FirstClass.y)

#ouyput
   5
   5
   15
   test
```

Class general form

```
class FirstClass:
        x=5
        def setdata(self, value):
            self.x = value
        def display(self):
            print(self.x)


print(FirstClass.x)
FirstClass.display(FirstClass)

FirstClass.setdata(FirstClass,15)
FirstClass.display(FirstClass)

FirstClass.y='test'
print(FirstClass.y)

#ouyput
   5
   5
   15
   test
```

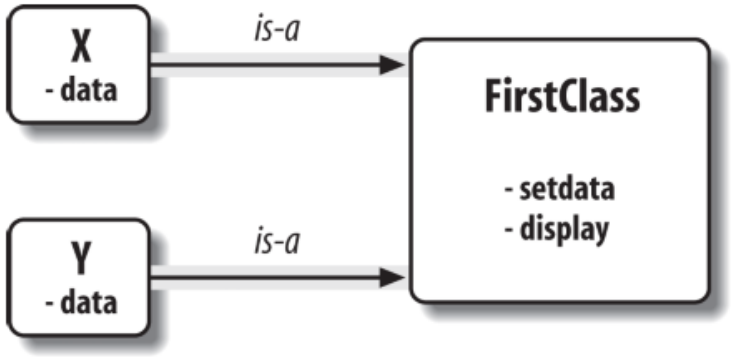
Class instantiation

```
class FirstClass:                       # Define a class object
        def setdata(self, value):       # Define class's methods
            self.data = value           # self is the instance
        def display(self):
            print(self.data)            # self.data: per instance

x = FirstClass()                    # Make two instances
y = FirstClass()                    # Each is a new namespace
x.setdata("King Arthur")            # Call methods: self is x
y.setdata(3.14159)                  # Runs: FirstClass.setdata(y, 3.14159)

x.display()                         # self.data differs in each instance
y.display()                         # Runs: FirstClass.display(y)


#output

   King Arthur
   3.14159
```

Class instantiation

```
class FirstClass:
        def setdata(self, value):
            self.data = value
        def display(self):
            print(self.data)

x = FirstClass()
y = FirstClass()
x.setdata("King Arthur")
y.setdata(3.14159)

x.display()
y.display()


#output

   King Arthur
   3.14159
```

Class instantiation



Class instantiation

```
class FirstClass:                       # Define a class object
        def setdata(self, value):       # Define class's methods
            self.data = value           # self is the instance
        def display(self):
            print(self.data)            # self.data: per instance

x = FirstClass()                    # Make two instances
y = FirstClass()                    # Each is a new namespace
x.setdata("King Arthur")            # Call methods: self is x
y.setdata(3.14159)                  # Runs: FirstClass.setdata(y, 3.14159)

x.display()                         # self.data differs in each instance
y.display()                         # Runs: FirstClass.display(y)

x.data = "New value"                # Can get/set attributes
x.display()                         # Outside the class too


#output
   King Arthur
   3.14159
   New value
```

Class instantiation

```
class FirstClass:
        def setdata(self, value):
            self.data = value
        def display(self):
            print(self.data)

x = FirstClass()
y = FirstClass()
x.setdata("King Arthur")
y.setdata(3.14159)

x.display()
y.display()

x.data = "New value"
x.display()


#output
   King Arthur
   3.14159
   New value
```

Class instantiation

```
class FirstClass:                       # Define a class object
        def setdata(self, value):       # Define class's methods
            self.data = value           # self is the instance
        def display(self):
            print(self.data)            # self.data: per instance

x = FirstClass()                    # Make two instances
y = FirstClass()                    # Each is a new namespace
x.setdata("King Arthur")            # Call methods: self is x
y.setdata(3.14159)                  # Runs: FirstClass.setdata(y, 3.14159)

x.display()                         # self.data differs in each instance
y.display()                         # Runs: FirstClass.display(y)

x.data = "New value"                # Can get/set attributes
x.display()                         # Outside the class too

x.anothername = "spam"
print(x.anothername)

#output
   King Arthur
   3.14159
   New value
   spam
```

Class instantiation

```
class FirstClass:
        def setdata(self, value):
            self.data = value
        def display(self):
            print(self.data)

x = FirstClass()
y = FirstClass()
x.setdata("King Arthur")
y.setdata(3.14159)

x.display()
y.display()

x.data = "New value"
x.display()

x.anothername = "spam"
print(x.anothername)

#output
   King Arthur
   3.14159
   New value
   spam
```

Class instantiation

```
class FirstClass:                       # Define a class object
        def setdata(self, value):       # Define class's methods
            self.data = value           # self is the instance
        def display(self):
            print(self.data)            # self.data: per instance

x = FirstClass()                    # Make two instances
y = FirstClass()                    # Each is a new namespace
x.setdata("King Arthur")            # Call methods: self is x
y.setdata(3.14159)                  # Runs: FirstClass.setdata(y, 3.14159)

x.display()                         # self.data differs in each instance
y.display()                         # Runs: FirstClass.display(y)

x.data = "New value"                # Can get/set attributes
x.display()                         # Outside the class too

x.anothername = "spam"
print(x.anothername)
print(FirstClass.anothername)       # Error
```

Class instantiation

```
class FirstClass:
        def setdata(self, value):
            self.data = value
        def display(self):
            print(self.data)

x = FirstClass()
y = FirstClass()
x.setdata("King Arthur")
y.setdata(3.14159)

x.display()
y.display()

x.data = "New value"
x.display()

x.anothername = "spam"
print(x.anothername)
print(FirstClass.anothername)    # Error
```

Class instantiation

```
class FirstClass:                       # Define a class object
        def setdata(self, value):       # Define class's methods
            self.data = value           # self is the instance
        def display(self):
            print(self.data)            # self.data: per instance

x = FirstClass()                    # Make two instances
y = FirstClass()                    # Each is a new namespace
x.setdata("King Arthur")            # Call methods: self is x
y.setdata(3.14159)                  # Runs: FirstClass.setdata(y, 3.14159)

x.display()                         # self.data differs in each instance
y.display()                         # Runs: FirstClass.display(y)

x.data = "New value"                # Can get/set attributes
x.display()                         # Outside the class too

x.anothername = "spam"
print(x.anothername)

FirstClass.newv=7
print(FirstClass.newv)
print(x.newv)

#output

   King Arthur
   3.14159
   New value
   spam
   7
   7
```

Class instantiation

```
class FirstClass:
        def setdata(self, value):
            self.data = value
        def display(self):
            print(self.data)

x = FirstClass()
y = FirstClass()
x.setdata("King Arthur")
y.setdata(3.14159)

x.display()
y.display()

x.data = "New value"
x.display()

x.anothername = "spam"
print(x.anothername)

FirstClass.newv=7
print(FirstClass.newv)
print(x.newv)

#output

   King Arthur
   3.14159
   New value
   spam
   7
   7
```

```
class FirstClass:                    # Define a class object
        r=5
        def setdata(self, value): # Define class's methods
            self.x = value        # self is the instance
        def display(self):
            print(self.x)         # self.data: per instance

x = FirstClass()                 # Make two instances
y = FirstClass()                 # Each is a new namespace
print('x.r:', x.r)
print('FirstClass.r:', FirstClass.r)
x.x=98
print('x.x: ', end=' ')
x.display()
print('FirstClass.r:', FirstClass.r)
#print('FirstClass.x:', FirstClass.x)  # FF has no x

FirstClass.setdata(FirstClass,15)      # now FF has  x
print('FirstClass.x: ', end=' ')
FirstClass.display(FirstClass)
print('FirstClass.x:', FirstClass.x)
print('FirstClass.r:', FirstClass.r)


x.setdata("King Arthur")         # Call methods: self is x
y.setdata(3.14159)               # Runs: FirstClass.setdata(y, 3.14159)
print('x.x:', x.x)
print('y.x:', y.x)
print('FirstClass.x:', FirstClass.x)
```

```
x.r: 5
FirstClass.r: 5
x.x:  98
FirstClass.r: 5
FirstClass.x:  15
FirstClass.x: 15
FirstClass.r: 5
x.x: King Arthur
y.x: 3.14159
FirstClass.x: 15
```

```
class FirstClass:
        r=5
        def setdata(self, value):
            self.x = value
        def display(self):
            print(self.x)

x = FirstClass()
y = FirstClass()
print('x.r:', x.r)
print('FirstClass.r:', FirstClass.r)
x.x=98
print('x.x: ', end=' ')
x.display()
print('FirstClass.r:', FirstClass.r)
#print('FirstClass.x:', FirstClass.x)  # FF has no x

FirstClass.setdata(FirstClass,15)   # now FF has  x
print('FirstClass.x: ', end=' ')
FirstClass.display(FirstClass)
print('FirstClass.x:', FirstClass.x)
print('FirstClass.r:', FirstClass.r)


x.setdata("King Arthur")
y.setdata(3.14159)
print('x.x:', x.x)
print('y.x:', y.x)
print('FirstClass.x:', FirstClass.x)
```

```
x.x:  98
FirstClass.r: 5
FirstClass.x:  15
FirstClass.x: 15
FirstClass.r: 5
x.x: King Arthur
y.x: 3.14159
FirstClass.x: 15
```

```
class FirstClass:                    # Define a class object
        x=5
        def setdata(self, value): # Define class's methods
            self.x = value        # self is the instance
        def display(self):
            print(self.x)         # self.data: per instance

x = FirstClass()                  # Make two instances
y = FirstClass()                  # Each is a new namespace
print('x.x:', x.x)

print('FirstClass.x:', FirstClass.x)
x.x=x.x+98                    # x is local  of y
print('x.x: ', end=' ')
x.display()
print('FirstClass.x:', FirstClass.x)

FirstClass.setdata(FirstClass,15)
print('FirstClass.x: ', end=' ')
FirstClass.display(FirstClass)
print('FirstClass.x:', FirstClass.x)
print('x.x:', x.x)



x.setdata("King Arthur")          # Call methods: self is x
y.setdata(3.14159)                # Runs: FirstClass.setdata(y, 3.14159)
print('x.x:', x.x)
print('y.x:', y.x)
print('FirstClass.x:', FirstClass.x)
```

```
x.x: 5
FirstClass.x: 5
x.x:  103
FirstClass.x: 5
FirstClass.x:  15
FirstClass.x: 15
x.x: 103
x.x: King Arthur
y.x: 3.14159
FirstClass.x: 15
```

```
class FirstClass:
        x=5
        def setdata(self, value):
            self.x = value
        def display(self):
            print(self.x)

x = FirstClass()
y = FirstClass()
print('x.x:', x.x)

print('FirstClass.x:', FirstClass.x)
x.x=x.x+98
print('x.x: ', end=' ')
x.display()
print('FirstClass.x:', FirstClass.x)

FirstClass.setdata(FirstClass,15)
print('FirstClass.x: ', end=' ')
FirstClass.display(FirstClass)
print('FirstClass.x:', FirstClass.x)
print('x.x:', x.x)



x.setdata("King Arthur")
y.setdata(3.14159)
print('x.x:', x.x)
print('y.x:', y.x)
print('FirstClass.x:', FirstClass.x)
```

```
FirstClass.x: 5
x.x:  103
FirstClass.x: 5
FirstClass.x:  15
FirstClass.x: 15
x.x: 103
x.x: King Arthur
y.x: 3.14159
FirstClass.x: 15
```

Constructor

```
class FirstClass:                       # Define a class object
    def __init__(self, value):
        self.data = value
    def setdata(self, value):   # Define class's methods
        self.data = value       # self is the instance
    def display(self):
        print(self.data)        # self.data: per instance

x = FirstClass("King Arthur")     # Make two instances
y = FirstClass(3.14159)           # Each is a new namespace

x.display()                       # self.data differs in each instance
y.display()                       # Runs: FirstClass.display(y)

x.setdata("Second King")          # Call methods: self is x
y.setdata(3.14159159)             # Runs: FirstClass.setdata(y, 3.14159)

x.display()                       # self.data differs in each instance
y.display()                       # Runs: FirstClass.display(y)


#output
    King Arthur
    3.14159
    Second King
    3.14159159
```

Constructor

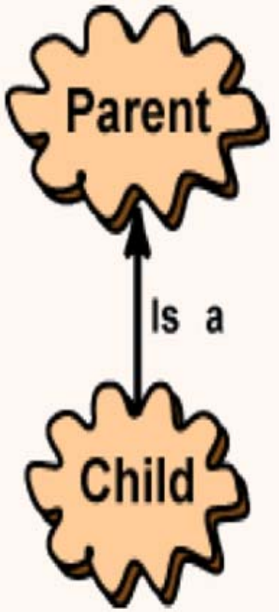
```
class FirstClass:
    def __init__(self, value):
        self.data = value
    def setdata(self, value):
        self.data = value
    def display(self):
        print(self.data)

x = FirstClass("King Arthur")
y = FirstClass(3.14159)

x.display()
y.display()

x.setdata("Second King")
y.setdata(3.14159159)

x.display()
y.display()


#output
    King Arthur
    3.14159
    Second King
    3.14159159
```

Inheritance

Object oriented languages have a feature called **inheritance**. Inheritance enables you to define a new class based upon an existing class. The new class is similar to the existing class, but has additional member variables and methods. This makes programming easier because you can build upon an existing class instead of starting out from scratch.

Inheritance

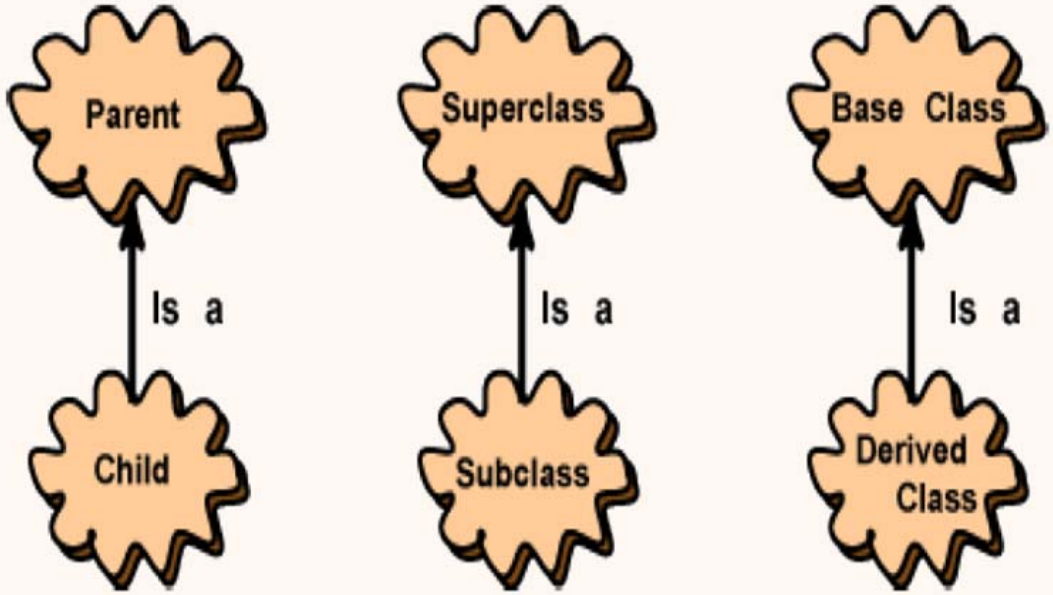
The class that is used to define a new class is called a **parent** class (or superclass or base class.) The class based on the parent class is called a **child** class (or subclass or derived class.)



In Java, (unlike with humans) children inherit characteristics from *just one* parent. This is called **single inheritance**. Some languages allow a child to inherit from more than one parent. This is called **multiple inheritance**. With multiple inheritance, it is sometimes hard to tell which parent contributed what characteristics to the child (as with humans). Java avoids these problems by using single inheritance.

Inheritance

There are three sets of phrases for describing inheritance relationships: parent/child, base class/derived class, superclass/subclass. Programmers use all three sets interchangebly.



Inheritance

```
class FirstClass:                    # Define a class object
    def setdata(self, value):        # Define class's methods
        self.data = value            # self is the instance
    def display(self):
        print(self.data)             # self.data: per instance


class SecondClass(FirstClass):       # Inherits setdata
    def display(self):               # Changes display
        print('Current value = "%s"' % self.data)

x = FirstClass()                     # Make two instances
z = SecondClass()
z.setdata(10)                        # Finds setdata in FirstClass
x.setdata(42)                        # Finds setdata in FirstClass
z.display()
x.display()                          # Outside the class too

#output
    Current value = "10"
    42
```

Inheritance

```
class FirstClass:
    def setdata(self, value):
        self.data = value
    def display(self):
        print(self.data)


class SecondClass(FirstClass):
    def display(self):
        print('Current value = "%s"' % self.data)

x = FirstClass()
z = SecondClass()
z.setdata(10)
x.setdata(42)
z.display()
x.display()

#output
    Current value = "10"
    42
```

Local attribute

```
class FirstClass:                  # Define a class object
    def setdata(self, value):      # Define class's methods
        self.data = value          # self is the instance
    def display(self):
        print(self.data)           # self.data: per instance


class SecondClass(FirstClass):     # Inherits setdata
    dataS=5
    def display(self):             # Changes display
        print(self.dataS)
        print('Current value = "%s"' % self.data)


x = FirstClass()                   # Make two instances
z = SecondClass()
z.setdata(10)                      # Finds setdata in FirstClass
x.setdata(42)                      # Finds setdata in FirstClass
z.display()
x.display()                        # Outside the class too

#output
    5
    Current value = "10"
    42
```

Local attribute

```
class FirstClass:
    def setdata(self, value):
        self.data = value
    def display(self):
        print(self.data)


class SecondClass(FirstClass):
    dataS=5
    def display(self):
        print(self.dataS)
        print('Current value = "%s"' % self.data)


x = FirstClass()
z = SecondClass()
z.setdata(10)
x.setdata(42)
z.display()
x.display()

#output
    5
    Current value = "10"
    42
```

Calling superclass methods

```
class FirstClass:                    # Define a class object
    def setdata(self, value):        # Define class's methods
        self.data = value            # self is the instance
    def display(self):
        print(self.data)             # self.data: per instance


class SecondClass(FirstClass):       # Inherits setdata
    dataS=5
    def display(self):               # Changes display
        print(self.dataS)
        print('Current value = "%s"' % self.data)
        FirstClass.display(self)


x = FirstClass()                     # Make two instances
z = SecondClass()
z.setdata(10)
x.setdata(42)
z.display()
x.display()

#output
    5
    Current value = "10"
    10
    42
```

Superclass Constructor

```
class FirstClass:                    # Define a class object
    def __init__(self, value):
         self.data = value
    def setdata(self, value):        # Define class's methods
        self.data = value            # self is the instance
    def display(self):
        print(self.data)             # self.data: per instance


class SecondClass(FirstClass):       # Inherits setdata
    dataS=5
    def display(self):               # Changes display
        print(self.dataS)
        print('Current value = "%s"' % self.data)


x = FirstClass(15)                   # Make two instances
z = SecondClass(17)
z.display()
x.display()

#output
    5
    Current value = "17"
    15
```

Superclass Constructor

```
class FirstClass:
    def __init__(self, value):
         self.data = value
    def setdata(self, value):
        self.data = value
    def display(self):
        print(self.data)


class SecondClass(FirstClass):
    dataS=5
    def display(self):
        print(self.dataS)
        print('Current value = "%s"' % self.data)


x = FirstClass(15)
z = SecondClass(17)
z.display()
x.display()

#output
    5
    Current value = "17"
    15
```

Subclass Constructor

```
class FirstClass:                  # Define a class object
    def __init__(self, value):
        self.data = value
    def setdata(self, value):      # Define class's methods
        self.data = value          # self is the instance
    def display(self):
        print('Data in first class: ', self.data)


class SecondClass(FirstClass):     # Inherits setdata
    dataS=5
    def __init__(self, value):
        self.dataS = value
    def display(self):             # Changes display
        print('Data from Superclass', self.data)
        print('Current value = "%s"' % self.dataS)



x = FirstClass(10)
z = SecondClass(20)
z.setdata(10)
x.setdata(42)
z.display()
x.display()

#output
    Data from Superclass 10
    Current value = "20"
    Data in first class:  42
```

Subclass Constructor

```
class FirstClass:
    def __init__(self, value):
        self.data = value
    def setdata(self, value):
        self.data = value
    def display(self):
        print('Data in first class: ', self.data)


class SecondClass(FirstClass):
    dataS=5
    def __init__(self, value):
         self.dataS = value
    def display(self):
        print('Data from Superclass', self.data)
        print('Current value = "%s"' % self.dataS)



x = FirstClass(10)
z = SecondClass(20)
z.setdata(10)
x.setdata(42)
z.display()
x.display()

#output
    Data from Superclass 10
    Current value = "20"
    Data in first class:  42
```

Calling Superclass Constructor

```
class FirstClass:                # Define a class object
   def __init__(self, value):
      self.data = value
   def setdata(self, value):  # Define class's methods
      self.data = value        # self is the instance
   def display(self):
      print('Data in first class: ', self.data)


class SecondClass(FirstClass):  # Inherits setdata
   dataS=5
   def __init__(self, value, valueS):
      FirstClass. __init__(self, value)
      self.dataS = valueS
   def display(self):            # Changes display
      print('Data from Superclass', self.data)
      print('Current value = "%s"' % self.dataS)



x = FirstClass(10)
z = SecondClass(20, 30)
z.display()
x.display()
z.setdata(10)
z.display()

#output

Data from Superclass 20
Current value = "30"
Data in first class:  10
Data from Superclass 10
Current value = "30"
```

Calling Superclass Constructor

```
class FirstClass:
    def __init__(self, value):
        self.data = value
    def setdata(self, value):
        self.data = value
    def display(self):
        print('Data in first class: ', self.data)


class SecondClass(FirstClass):
    dataS=5
    def __init__(self, value, valueS):
        FirstClass. __init__(self, value)
        self.dataS = valueS
    def display(self):
        print('Data from Superclass', self.data)
        print('Current value = "%s"' % self.dataS)



x = FirstClass(10)
z = SecondClass(20, 30)
z.display()
x.display()
z.setdata(10)
z.display()

#output

Data from Superclass 20
Current value = "30"
Data in first class:  10
Data from Superclass 10
Current value = "30"
```

Encapsulation

- Encapsulation is placing the data and the functions that work on that data in the same place. While working with procedural languages, it is not always clear which functions work on which variables but object-oriented programming provides you framework to place the data and the relevant functions together in the same object.
- Also we should be able to hide the details of class form outside word.

Encapsulation

- For this, the members of a class can be defined as

Name	Notation	Behaviour
name	Public	Can be accessed from inside and outside
_name	Protected	Like a public member, but they shouldn't be directly accessed from outside.
__name	Private	Can't be seen and accessed from outside

Encapsulation

- If an identifier doesn't start with an underscore character "_" it can be accessed from outside, i.e. the value can be read and changed. Data can be protected by making members private or protected. Instance variable names starting with two underscore characters cannot be accessed from outside of the class. At least not directly, but they can be accessed through private name mangling. That means, private data __A can be accessed by the following name construct: instance_name._classname__A.

Encapsulation

- If an identifier is only preceded by one underscore character, it is a protected member. Protected members can be accessed like public members from outside of class.

Encapsulation

```
class Encapsulation():
    def __init__(self, a, b, c):
        self.public = a
        self._protected = b
        self.__private = c

x = Encapsulation(11,13,17)
print(x.public)
x._protected=23
print(x._protected)
print(x.__private)
```

```
11
23
Traceback (most recent call last):
  File "C:/Documents and Settings/admin/Desktop/intro-python/example
py", line 11, in <module>
    print(x.__private)
AttributeError: 'Encapsulation' object has no attribute '__private'
```

Encapsulation

```
class Person():
    def __init__(self):
         self.A = 'Yang Li'
         self.__B = 'Yingying Gu'
    def printName(self):
        print(self.A)
        print(self.__B)

P=Person()
print(P.A)
#print(P.__B)
P.printName()

#output

Yang Li
Yang Li
Yingying Gu
```