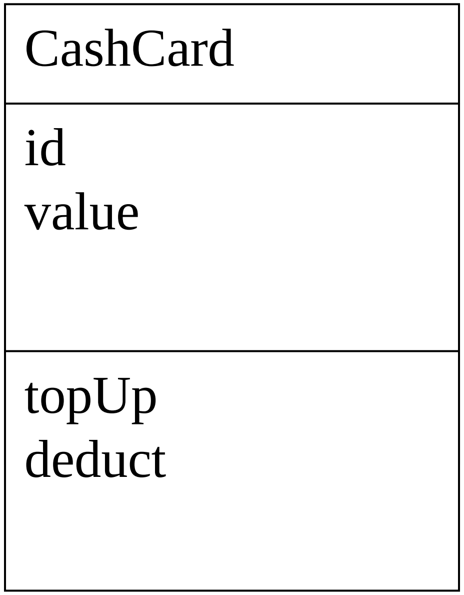
ICT162

Object Oriented Programming

Seminar 3 Inheritance

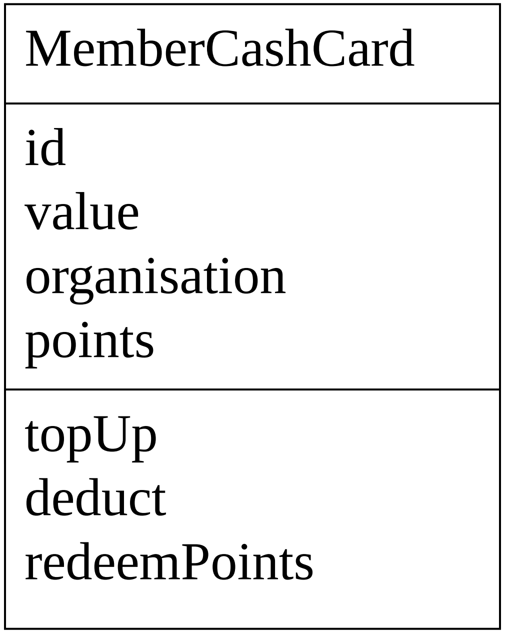
Introduction to Inheritance



CashCard

id
value

topUp
deduct



MemberCashCard

id
value
organisation
points

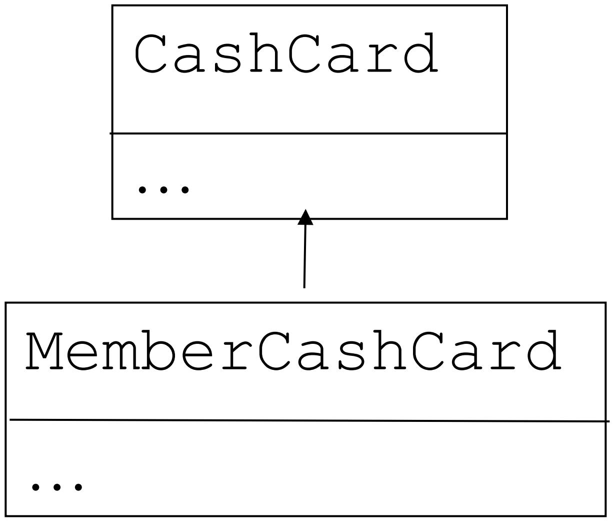
topUp
deduct
redeemPoints

- Member Cash Cards are issued by organisations (e.g NTUC)
- Accumulates points for purchases with the organisation

Inheritance

- A class can inherit the attributes and methods from another class.

E.g. the MemberCashCard class can inherit from CashCard class



Avoid writing separate classes with similar attributes and methods

Inheritance

- MemberCashCard inherits ALL the attributes of CashCard
  - id and value
- It can have additional attributes
  - Organisation and points

- MemberCashCard inherits ALL the methods of the CashCard
- It contains additional methods

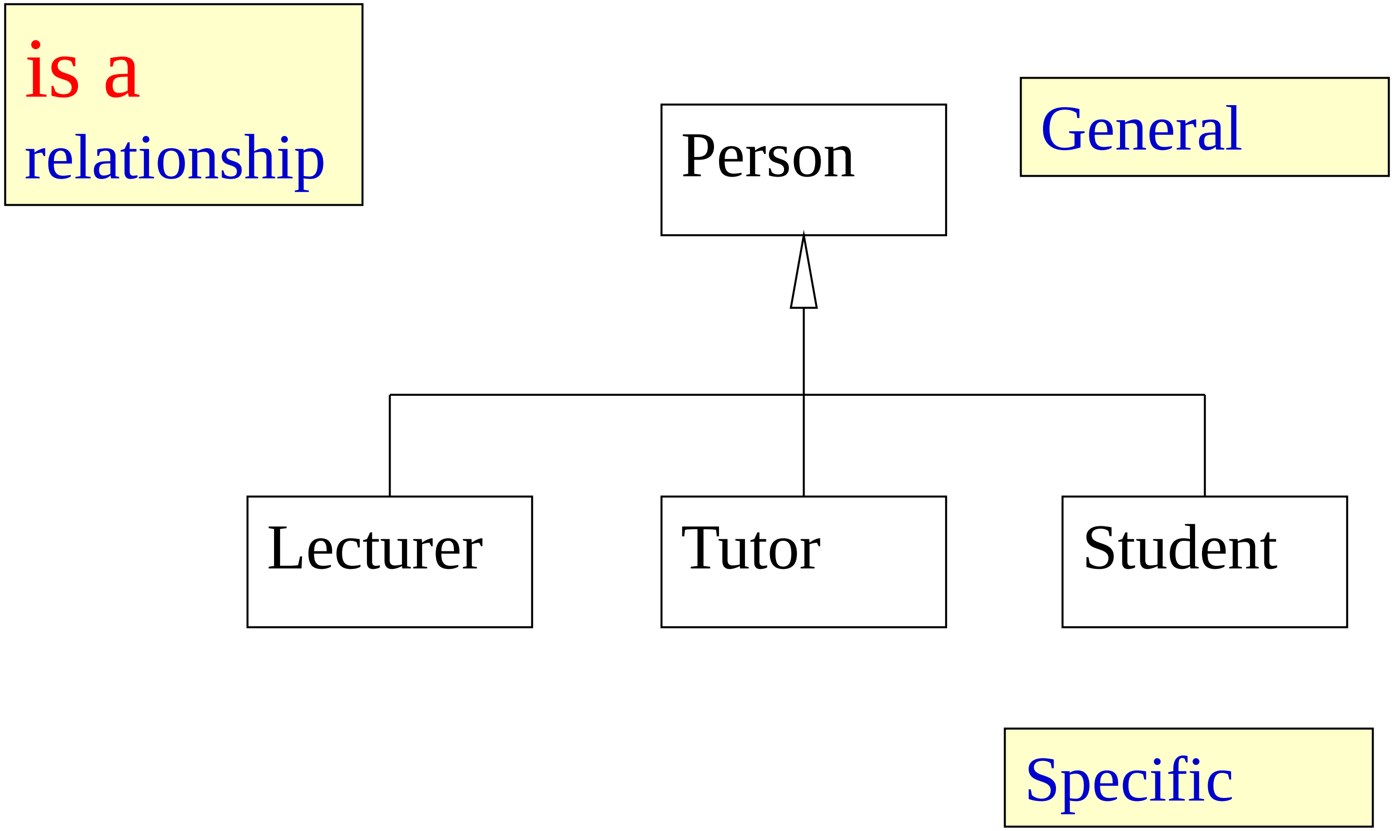
Inheritance

- MemberCashCard inherits from CashCard
  - CashCard - Superclass or Base class or Parent class
  - MemberCashCard - Subclass or Child class

- **Is-a** relationship

  MemberCashCard “is-a” CashCard

Inheritance – is a relationship

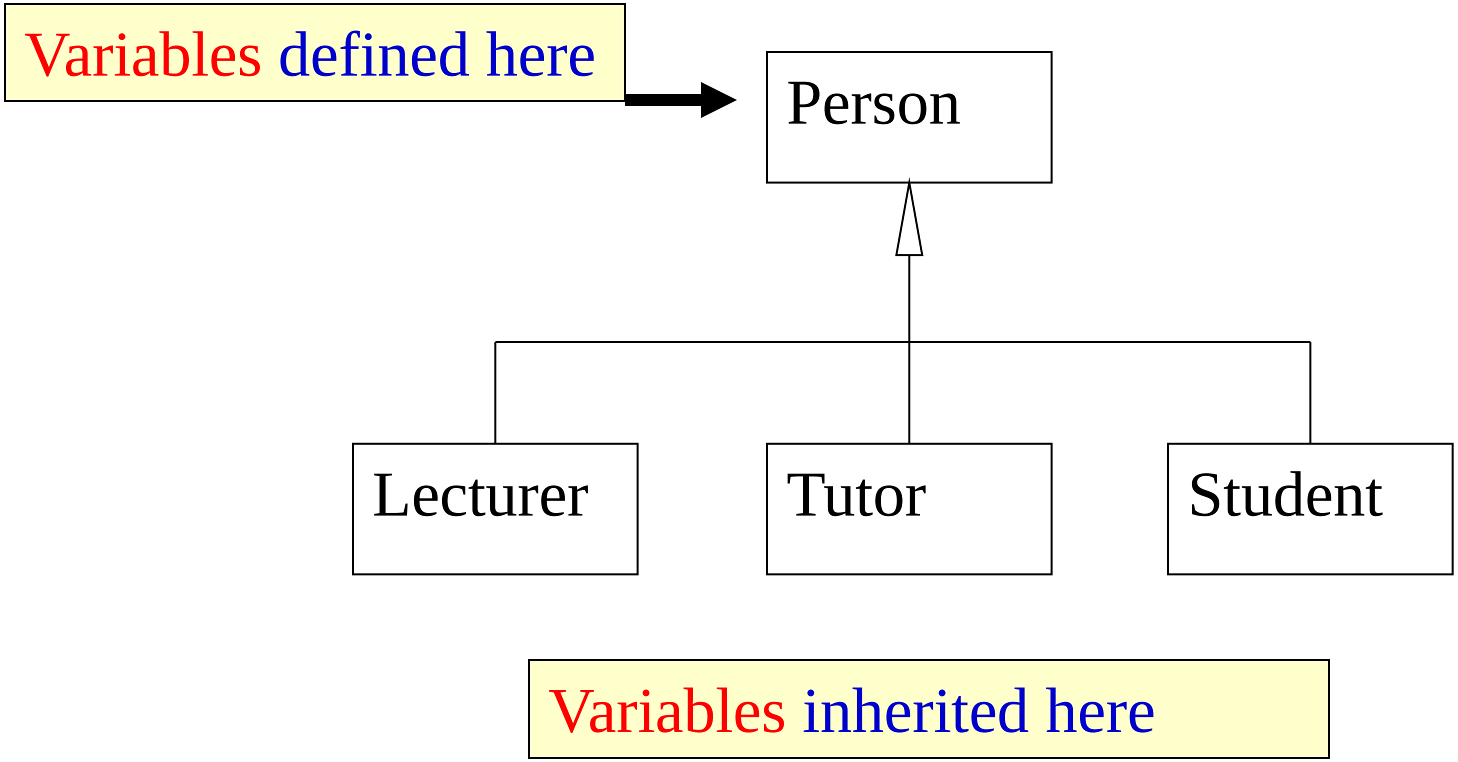


is a relationship

General

Specific

Inheritance



Variables defined here

Person

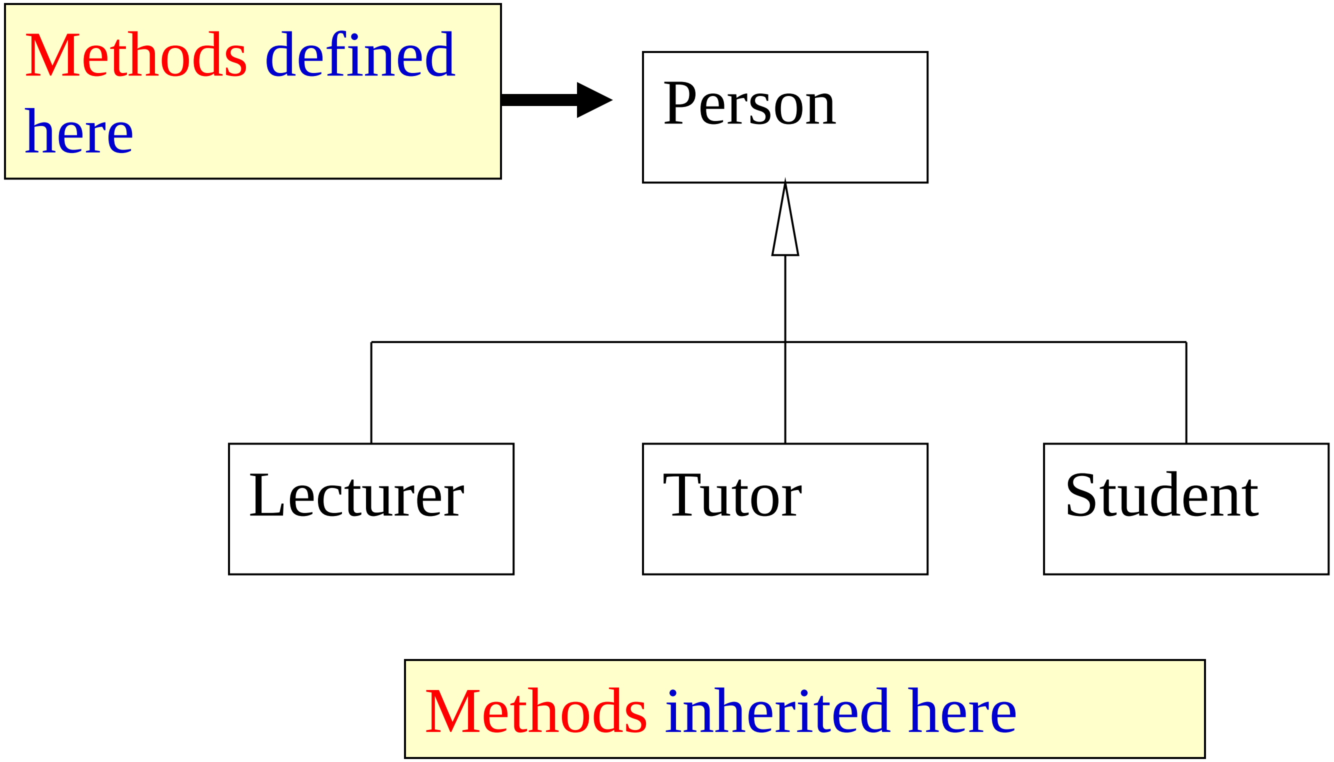
Lecturer

Tutor

Student

Variables inherited here

Inheritance



Methods defined here

Person

Lecturer

Tutor

Student

Methods inherited here

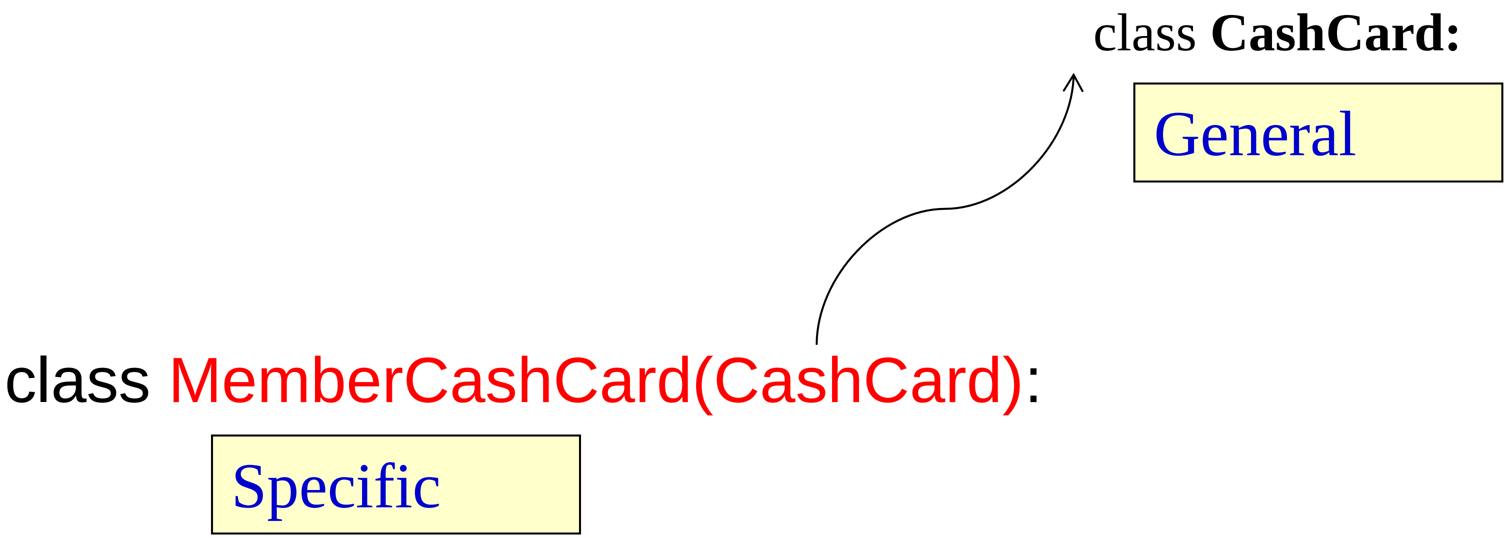
Composition vs Inheritance

- Inheritance
  - Is-a relationship
  - Inherits all the superclass’s instance variables and methods
  - No need to re-invent the wheel
- Composition
  - Has-a relationship
  - Has control over the object as able to invoke the methods of the object

```
class CashCard:
    _bonusRate = 0.01
    _bonusAmount = 100
   def __init__(self, id, amount):
       self._id = id
       self._balance = amount
       self.addBonus(amount)


      def addBonus(self, amount):
       if amount >= type(self)._bonusAmount :
           self._balance += amount * type(self)._bonusRate


     @property
     def id(self):
         return self._id
```

```
   @property
    def balance(self):
        return self._balance

    def deduct(self, amount):
        if self._balance >= amount:
            self._balance -= amount
            return True
       return False

    def topUp(self, amount):
       if amount > 0:
           self._balance += amount
           self.addBonus(amount)

    def __str__(self):
       return 'Id: {} Balance: ${:.2f}'.format(self._id, self._balance)
```

MemberCashCard class



class **CashCard:**

General

class MemberCashCard(CashCard):

Specific

MemberCashCard class

```
class CashCard:
    _bonusfor100 = 0.01
    def __init__(self, id, amount):
        self._id = id
        self._balance = amount
        self.addBonus(amount
```

```
class MemberCashCard(CashCard):
    def __init__ (self, organisation, id, amount):
        super().__init__(id, amount)
        self._organisation = organisation
        self._points = 0
```

Other methods

```
class MemberCashCard(CashCard):
  …

  def redeemPoints(self, pts):
      if self._points >- pts:
       self._points -=  pts
  def deduct(self, amt):
      if super().deduct(amt):
          self._points  += int(amt)
          return True
      return False
```

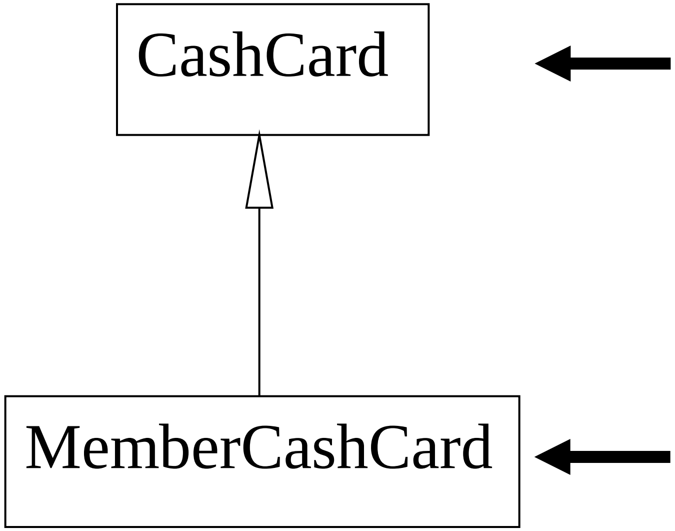
```
class CashCard:
   …
    def deduct(self, amount):
        if self._balance >= amount:
            self._balance -= amount
            return True
        return False
```

Method Overriding

- Method overriding happens when the subclass and superclass have the same method name and parameters
  - The deduct(amt) method in MemberCashCard class overrides the deduct(amt) method in CashCard class.

- Method overriding by refinement
  - With super().deduct(amt) in the method
- Method overriding by replacement
  - Without super().deduct(amt) in the method

Overriding by Refinement



CashCard

MemberCashCard

```
def deduct(amt):
          ....
```

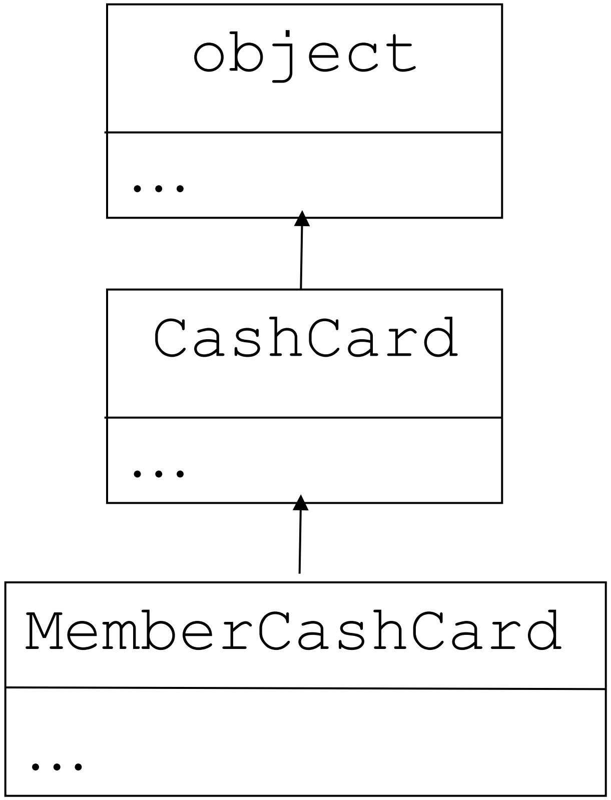
```
def deduct(amt):
     if super().deduct(amt):
          ...
```

signature must be the same and subclass method calls the superclass’s method – overriding by refinement

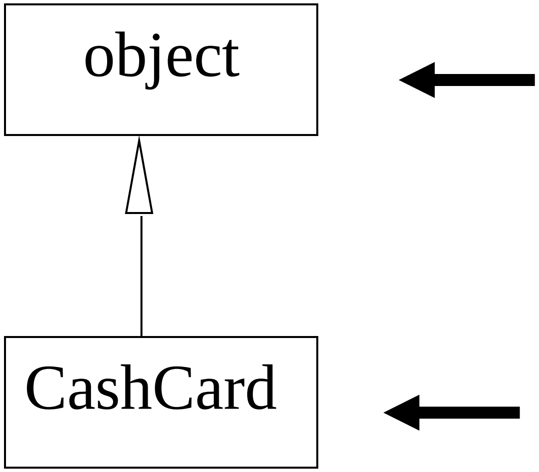
```
class MemberCashCard(CashCard):
    def __init__ (self, organisation, id, amount):
        super().__init__(id, amount)
        self._organisation = organisation
        self._points = 0

    def redeemPoints(self, pts):
        if self._points >= pts:
            self._points -=  pts

    def deduct(self, amt):
        if super().deduct(amt):
            self._points  += int(amt)
            return True
        return False

    def __str__(self):
        return super().__str__() + ' Organisation: {:10} Points:{}'.\
format(self._organisation, self._points)
```

The object class

- All classes inherit from the object class
  - A base for all classes.
- Instances are new featureless object.
- It has the methods that are common to all instances of Python classes.



Overriding by Replacement

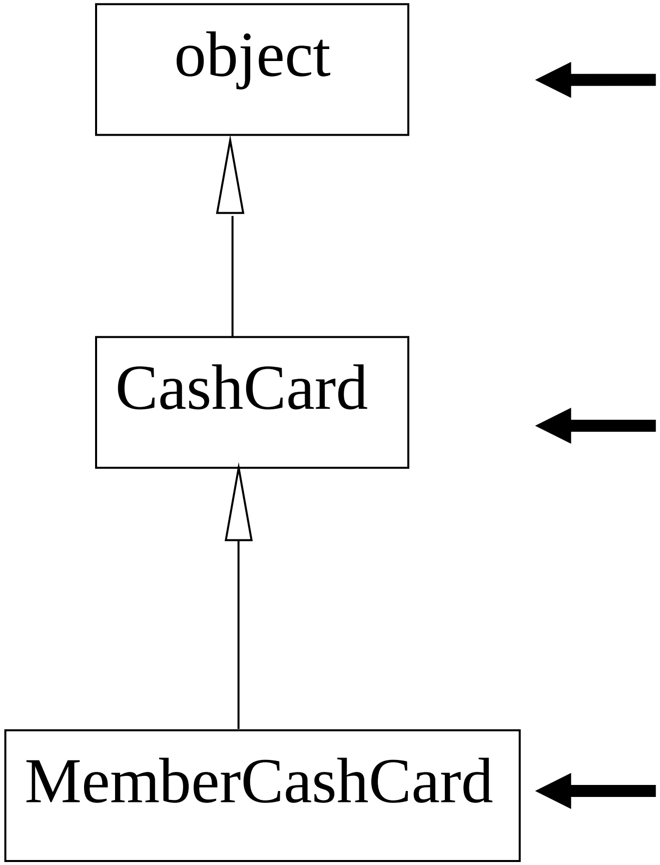


```
def __str__(self):
        # returns hash code
```

```
def __str__(self):
        return 'Id: {} Balance:
${:.2f}'.format(self._id, self._balance)
```

The __str__ method in CashCard class overrides the __str__ method in the object class by replacement.

Overriding by Replacement /Refinement



object

CashCard

MemberCashCard

```
def __str__(self):
      # returns hash code
```

```
def __str__(self):
      return 'Id: {} Balance:
${:.2f}'.format(self._id, self._balance)
```

```
def __str__(self):
      return super().__str__() + '
Organisation: {:10} Points:{}'.\
format(self._organisation, self._points))
```

Duck Typing in Python

```
c = MemberCashCard('SUSS', ‘1’, 500)
c.deduct(100)

c = CashCard(‘2’, 300))
c.deduct(100)
```

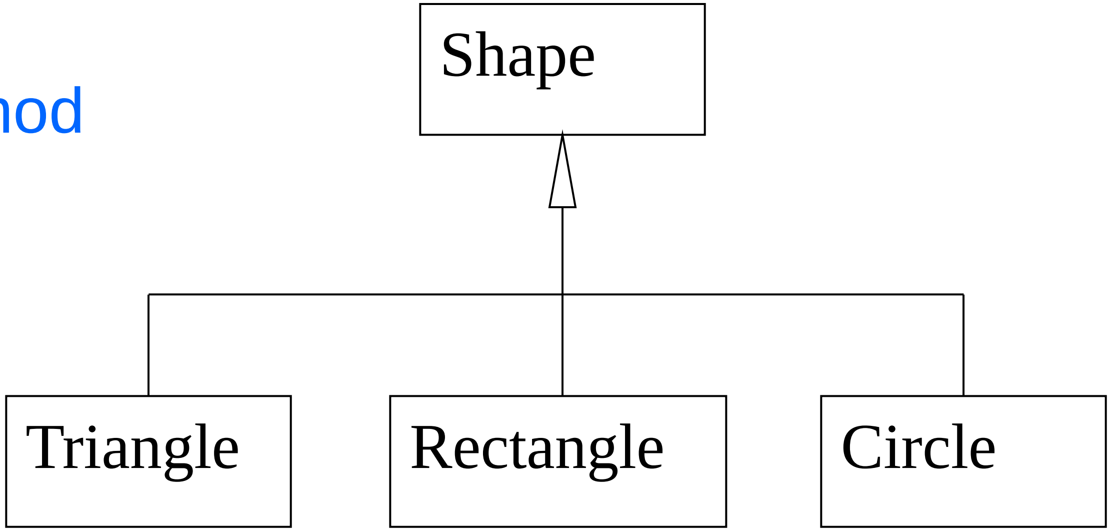
Abstract Class

- Up the inheritance hierarchy, classes get more and more general
  - classes act as a framework for other classes

- Abstract classes
  - may contain method headers with no implementation (pass statement)
  - No objects can be instantiated from these classes

Abstract Class

```
from abc import ABC, abstractmethod
class Shape(ABC):

    @abstractmethod
    def area(self):
        pass
```



implement different area( ) here

Abstract Shape class (superclass)

```
from abc import ABC, abstractmethod
class Shape(ABC):

    def __init__(self, length):
        self._length = length

    @property
    def length(self):
        return self._length

    @abstractmethod
    def area(self):
        pass
```

Rectangle class (subclass of Shape)

```
class Rectangle(Shape):
    def __init__(self, length, width):
        super().__init__(length)
        self._width = width

    @property
    def width(self):
        return self._width

    def area(self):
        return self._length * self._width
```

Circle class (subclass of Shape)

```
from math import pi
class Circle(Shape):
    def __init__(self, radius):
        super().__init__(radius)

    @property
    def radius(self):
        return self.length

    def area(self):
        return pi * self.radius ** 2
```

Triangle class (subclass of Shape)

```
class Triangle(Shape):
    def __init__(self, base, height):
        super().__init__(base)
        self._height = height

    @property
    def base(self):
        return self.length

    @property
    def height(self):
        return self._height

    @height.setter
    def height(self, height):
        self._height = height

    def area(self):
        return 0.5 * self.base * self._height
```

Creating Shape objects

```
def main():
    shapes = []
    shapes.append(Circle(2))
    shapes.append(Rectangle(2, 10))
    shapes.append(Triangle(10, 5))
    shapes.append(Circle(3))
    shapes.append(Triangle(5, 7))
    shapes.append(Circle(4))

    for s in shapes:
        print(s.area())
```

Polymorphism

- A Greek word that means “many forms”

```
for s in shapes:
        print(s.area())
```

The variable s is a polymorphic variable as it can take on the form of a circle or rectangle object.

The statement s.area() is polymorphic as its meaning depends on what s is.

Using default parameters

```
class CashCard:
    def __init__(self, id, amount = None):
        self._id = id
        if amount is None:
            amount = 20
        self._balance = amount
        self.addBonus(amount)
```

```
c1 = CashCard(1)
c2 = CashCard(2, 10)
```

Using default parameters

```
class MemberCashCard(CashCard):
    def __init__ (self,organisation, id, amount = None):
        super().__init__(id, amount)
        self._organisation = organisation
        self._points = 0
```

```
m1= MemberCashCard(‘NTUC’, 1)
m2 = MemberCashCard(‘SUSS’, 2, 20)
```

Using default parameters

```
class CashCard:
    def deduct(self, amount = None):
        if amount is None:
            amount = 10
        if self._balance >= amount:
            self._balance -= amount
            return True
        return False
```

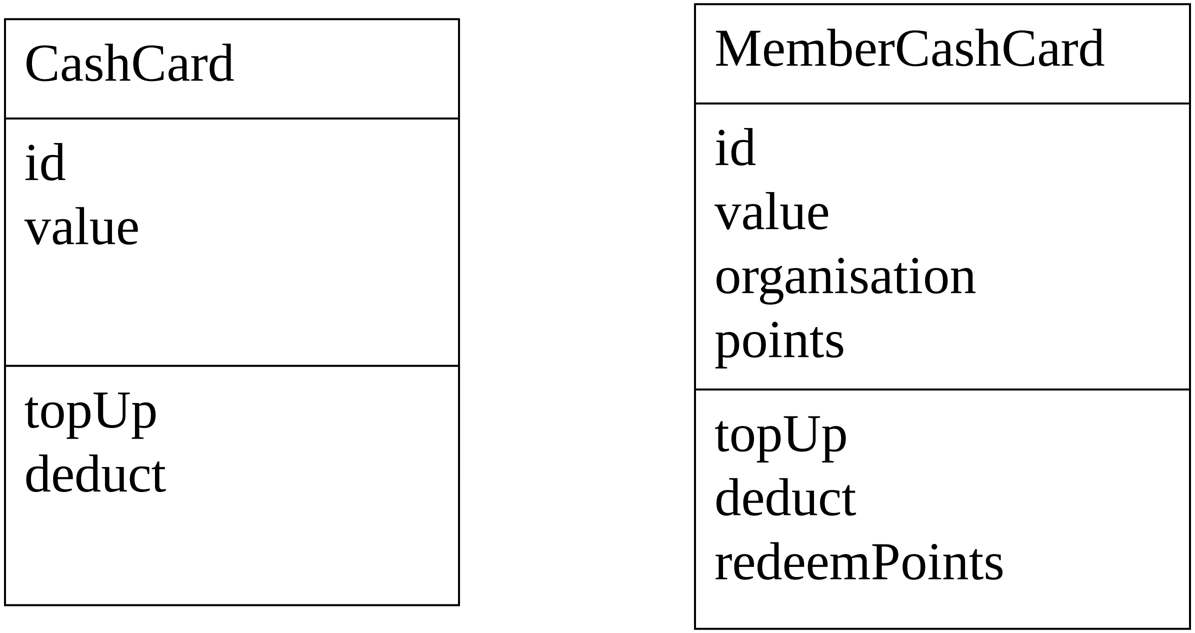
```
c1.deduct(20)
c1.deduct()
```

Using default parameters

```
class MemberCashCard(CashCard):
    def deduct(self, amt = None):
        if amt is None:
            amt = 10
        if super().deduct(amt):
            self._points  += int(amt)
            return True
        return False
```

```
m1.deduct(20)
m1.deduct()
```

Adding objects in the inheritance hierarchy (concrete superclass)



CashCard

id
value

topUp
deduct

MemberCashCard

id
value
organisation
points

topUp
deduct
redeemPoints

```
cards= []

cards.append(CashCard(‘1’, 20))
cards.append(MemberCashCard(”ntuc”,“2”,10))
```

Adding objects in the inheritance hierarchy (concrete superclass)

- Are the following ok?
  - cards[i].deduct(10); # ok ?

    # ok when cards[i] is a MemberCashCard object?

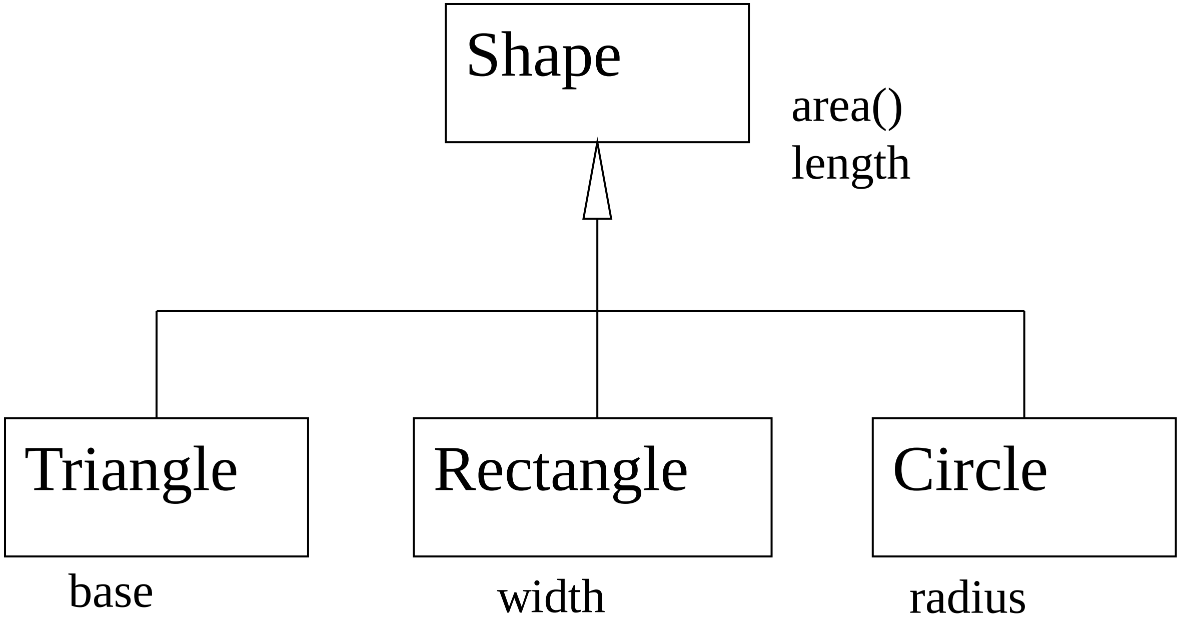
    # ok when cards[i] is a CashCard object ?

  - cards[i].redeemPoints(100) # ok?

    # ok when cards[i] is a MemberCashCard object?

    # ok when cards[i] is a CashCard object ?

Adding objects in the inheritance hierarchy (abstract superclass)



```
shapes = []

shapes.append(Circle(2))
shapes.append(Rectangle(3, 5))
shapes,append(Triangle(4,8))
```

Adding objects in the inheritance hierarchy (abstract superclass)

Are the following ok?

- shapes[i].area() // ok ?
- print(shapes[i].length) // ok ?
- print(shapes[i].base) // ok ?

# ok when shapes[i] is a Circle object?

# ok when shapes[i] is a Rectangle object ?

# ok when shapes[i] is a Triangle object ?