ICT162

Object Oriented Programming

Seminar 1 Class and Objects

Object Oriented Programming

- Programming that models after real life situations

- Put all related variables and methods together (Abstraction)

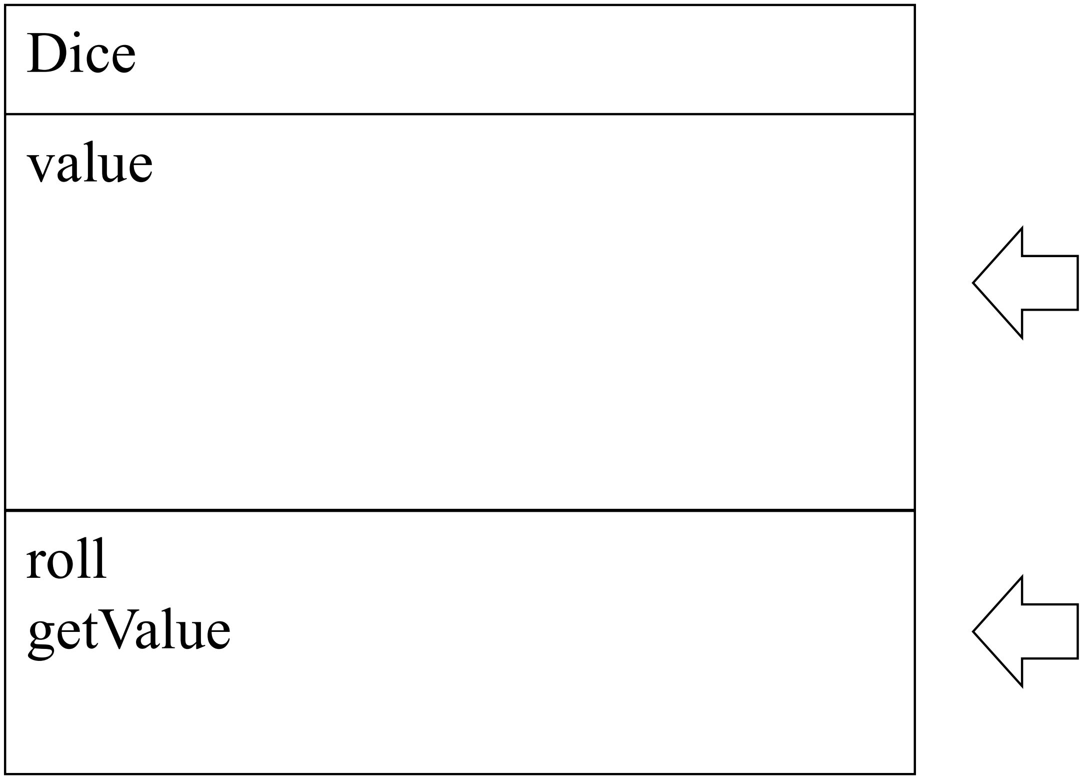
- Hides all details. Expose only through method call (Encapsulation)

Object Oriented Programming

- Class
  - A structure where we define all related variables belonging to an entity.
  - All related methods that process the variables
  - Only a template, actual object not created yet

- Objects or instances are ***actual*** entities
  - Object = identity + instance variables + methods

Basic Structure of a Class



Dice

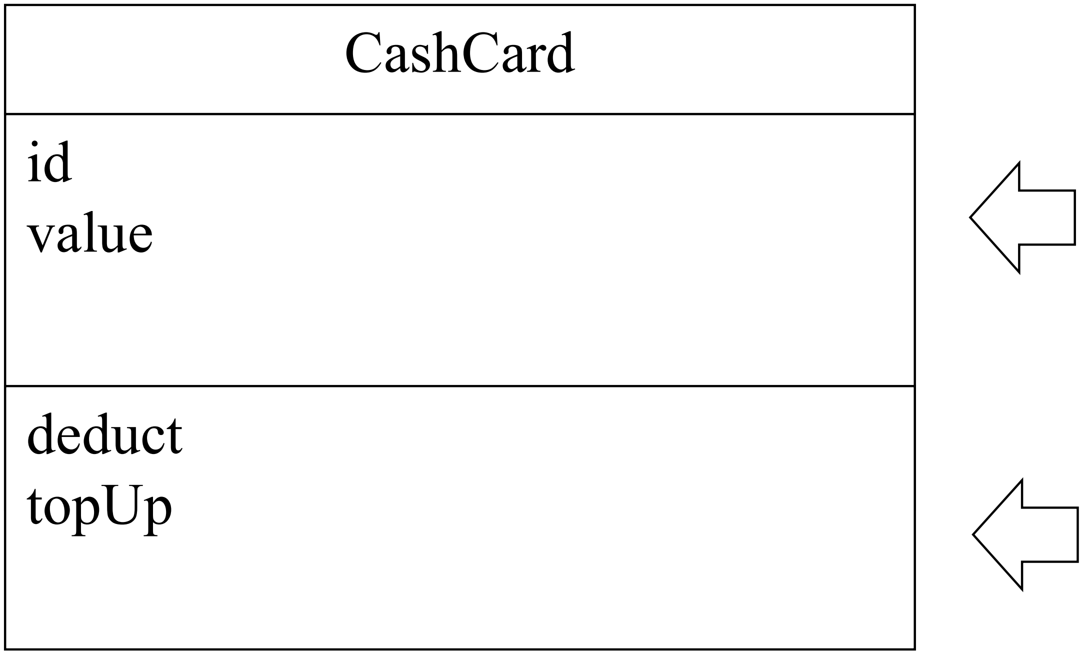
value

roll
getValue

Attributes, properties, characteristics, description

Capabilities, services, behaviour, functions, operations

Another Example



CashCard

id
value

deduct
topUp

Attributes, properties, characteristics, description

Capabilities, services, behaviour, functions, operations

Writing a Class

class className**:**

constructor initializes the values of instance variables

accessor or getter methods

mutator or setter methods

other methods

Constructor and Instance Variables

```
from random import randint
class Dice:
     def __init__(self):
          self.__value = randint(1,6)
```

- __init__ is the name for the constructor
- initializes the values of instance variables

  Note: Include only attributes relevant to the application
- __ (double underscore or dunder)

  private or hidden outside the class definition

Constructor and Instance Variables

```
class CashCard:
    def __init__(self, id, amount):
        self.__id = id
        self.__balance = amount
```

Creating objects

```
d1 = Dice()
d2 = Dice()
```

```
class Dice:
    def __init__(self):
```

```
c1 = CashCard('123', 20.0)
c2 = CashCard('456', 10.0)
```

```
class CashCard:
    def __init__(self, id, amount):
```

In the same order as the constructor parameters

Accessor or Getter methods

```
from random import randint
class Dice:
    def __init__(self):
        self.__value = randint(1,6)

    @property
    def value(self):
        return self.__value
```

Accessor or Getter methods

```
class CashCard:
    def __init__(self, id, amount):
        self.__id = id
        self.__balance = amount

    @property
    def id(self):
        return self.__id


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

Mutator or Setter methods

```
from random import randint
class Dice:
    def __init__(self):
        self.__value = randint(1,6)

    @property
    def value(self):
        return self.__value

    @value.setter
    def value(self, newValue):
        self.__value = newValue
```

It is unlikely that a Dice object has this setter method though!!!

Mutator or Setter methods

```
class CashCard:
    def __init__(self, id, amount):
        self.__id = id
        self.__balance = amount

    @property
    def id(self):
        return self.__id

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

Caution!!!!
These setter methods
are not a behavior of a
CashCard object

```
@id.setter
def id(self, newId):
    self.__id = newId

@balance.setter
def balance(self, newBalance):
    self.__balance = newBalance
```

Calling accessor and mutator methods

```
print(d1.value, d2.value)
```

```
d1.value = 50
```

```
@property
def value(self):
    return self.__value

@value.setter
def value(self, newValue):
    self.__value = newValue
```

Calling accessor and mutator methods

```
print(c1.id, c2.id)

print(c1.balance, c2.balance)
```

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

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

```
c1.id = '878’

c2.balance = 100
```

```
@id.setter
def id(self, newId):
    self.__id = newId

@balance.setter
def balance(self, newBalance):
    self.__balance = newBalance
```

Other methods - Behaviour

```
from random import randint
class Dice:
    def __init__(self):
        self.__value = randint(1,6)

    @property
    def value(self):
        return self.__value

    def roll(self):
        self.__value = randint(1,6)

    def __str__(self):
        return 'Value: {}'.format(self.__value)
```

Other methods - Behaviour

```
class CashCard:
    def __init__(self, id, amount):
        self.__id = id
        self.__balance = amount

    @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

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

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

Usually returns the attribute values as a str

Sending message to object

Format: object.*message*(parameters);

```
aDice = Dice()

aDice.roll()
print(aDice.value)
```

```
myCard = CashCard(“123”, 10.0)

myCard.deduct(2.5)
myCard.topUp(10.0)
print(myCard.balance)
```

Calling __str__ method

Rather than

```
print(aDice.__str__())
```

Simply

```
print(aDice)   or
print(str(aDice))  for string operation
```

Collection of Objects

```
cards = []
cards.append(CashCard(“11”, 10))
cards.append(CashCard(“12”, 20))
cards.append(CashCard(“13”, 30))
…

for c in cards:
    print(c.getBalance())
```

Method overloading – Default parameters

```
class CashCard:
    def __init__(self, id, amount = 20):
        self.__id = id
        self.__balance = amount

    def deduct(self, amount = 5):
        if self.__balance >= amount:
            self.__balance -= amount

    def topUp(self, amount=10):
        if amount > 0:
            self.__balance += amount
```

```
c1 = CashCard("123", 10.0)
c2 = CashCard("124")

c1.deduct(2.5)
c1.deduct()

c1.topUp(5)
c1.topUp()
```

Class variables

- Class variables
  - variables defined in a class outside methods
  - There is only 1 copy of this variable during execution versus the many copies of instance variables for every object instantiated
- For example, the Dice class records the number of sides its object has.

Class Variables and Methods

```
from random import randint
class Dice:
    __sides = 6

    @classmethod
    def getSides(cls):
        return cls.__sides

    @classmethod
    def setSides(cls, sides):
        cls.__sides = sides

    def __init__(self):
        self.__value = randint(1, type(self).getSides())
```

To get _sides:
Dice.getSides()

To set _sides:
Dice.setSides(10)

```
    @property
    def value(self):
        return self.__value

    def roll(self):
        self.__value = randint(1,\
                       type(self).getSides())

    def __str__(self):
         return 'Value: {}'.format \
                           (self.__value)
```

Class variables

- For a top up amount of 100 dollars or more, the cash card has additional 1% in value.

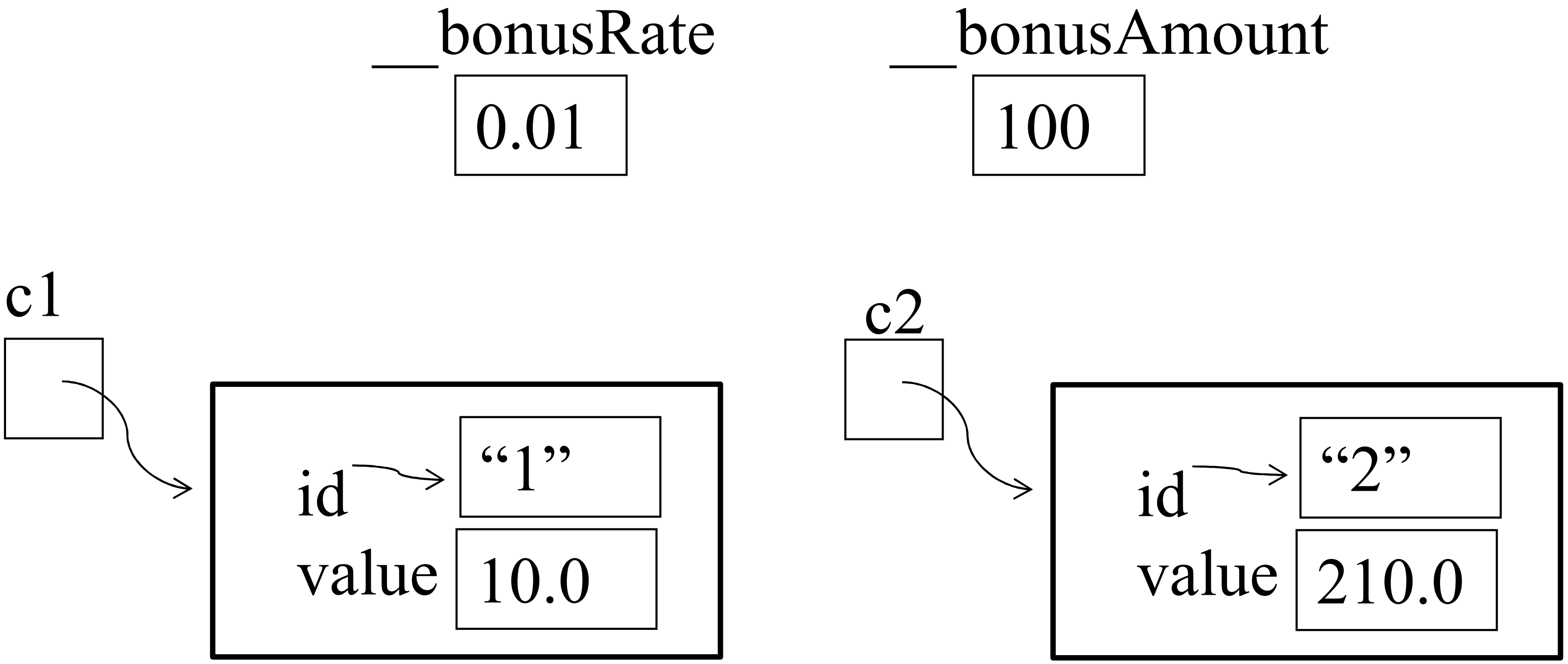
- 1% applies to top ups for all cash card
  - should not be an instance variable of every CashCard object

```
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

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

Class variables

c1 = CashCard(“1”, 10.0)

c2 = CashCard(“2”, 200.0)


__bonusRate
0.01
__bonusAmount
100
c1
id
“1”
value
10.0
c2
id
“2”
value
210.0