Object Oriented Programming (OOP)

- Essential concepts
- Defining and using a class
- Class-wide members

Essential concepts

- What is a class?
- What is an object?
- Class diagrams

What is a class?

A class is a representation of a real-world entity

- Defines data, plus methods to work on that data
- You can hide data from external code, to enforce encapsulation

Domain classes

- Specific to your business domain
- E.g. BankAccount, Customer, Patient, MedicalRecord

Infrastructure classes

- Implement technical infrastructure layer
- E.g. NetworkConnection, AccountsDataAccess, IPAddress

Error classes

- Represent known types of error

What is an Object?

An object is an instance of a class

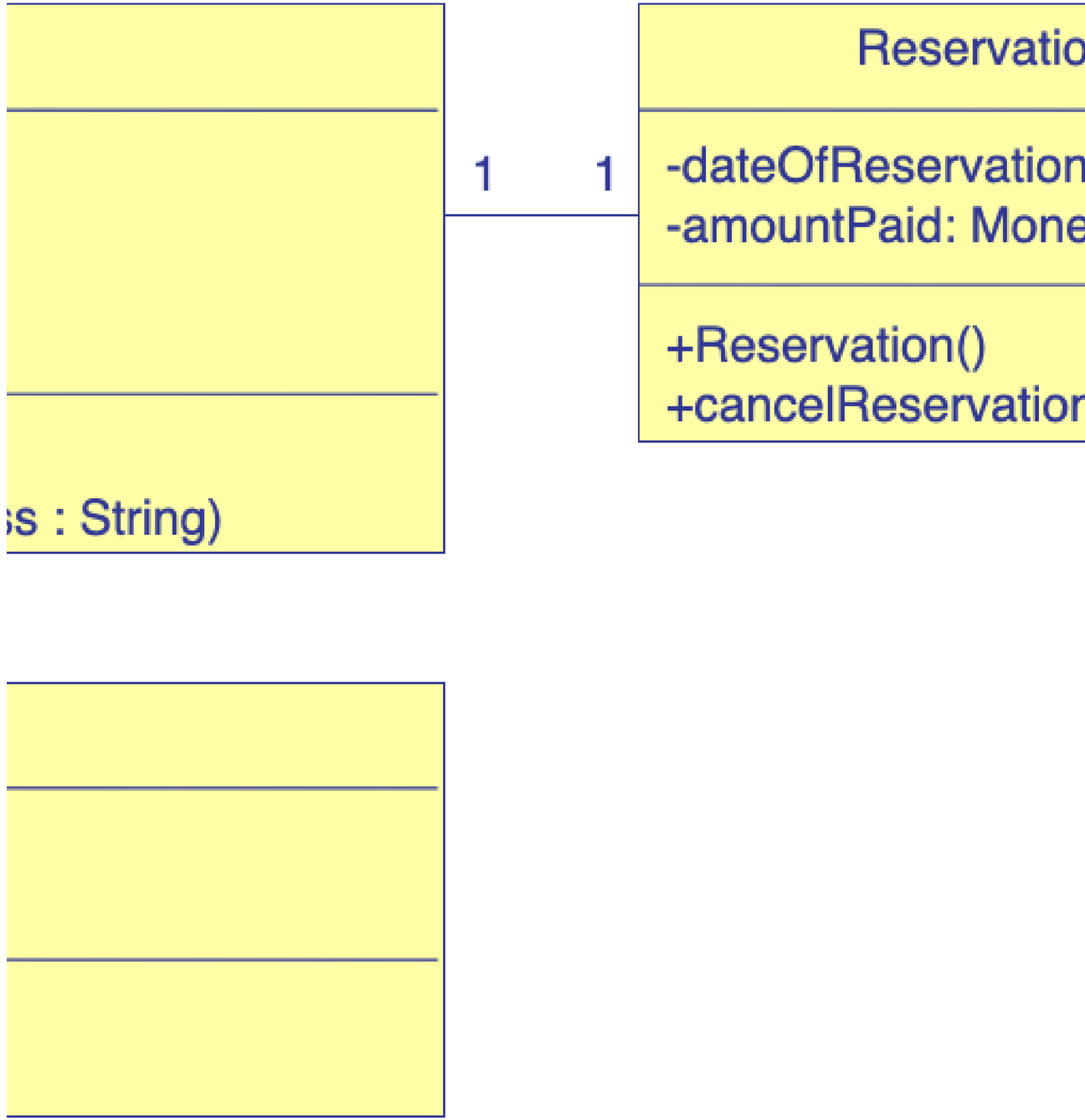
- Created (or "instantiated") by client code
- Each object is uniquely referenced by its memory address (no need for primary keys, as in a database)

Object management

- Objects are allocated on the garbage-collected heap
- An object remains allocated until the last remaining object reference disappears
- At this point, the object is available for garbage collection
- The garbage collector will reclaim its memory sometime thereafter

Class Diagrams

During OO analysis and design, you map the real world into candidate classes in your application.



Defining and using a class

- General syntax for class declarations
- Creating objects
- Defining and calling methods
- Defining instance variables
- Initialization methods
- Making an object’s attributes private
- Implementing method behaviour

General syntax for class declarations

General syntax for declaring a class:

```
class ClassName :
  #
  # Define attributes (data and methods) here.
  #
```

Example:

```
class BankAccount :
  #
  # Define BankAccount attributes (data and methods) here.
  #
```

Creating objects

To create an instance (object) of the class:

- Use the name of the class, followed by parentheses
- Pass initialization parameters if necessary (see later)
- You get back an object reference, which points to the object in memory

```
objectRef = ClassType(initializationParams)
```

Example

```
from accounting import BankAccount

acc1 = BankAccount()
acc2 = BankAccount()
```

Defining and calling methods

You can define methods in a class

- i.e. functions that operate on an instance of a class

In Python, methods must receive an extra first parameter

- Conventionally named self
- Allows the method to access attributes in the target object

```
class BankAccount :
  def deposit(self, amount):
    print("TODO: implement deposit() code")

  def withdraw(self, amount):
    print("TODO: implement withdraw() code")
```

Client code can call methods on an object

```
acc1 = BankAccount()
acc1.deposit(200)
acc1.withdraw(50)
```

Initialization methods (1/2)

You can implement a special method named `__init__()`

- Called automatically by Python, whenever a new object is created
- The ideal place for you to initialize the new object!
- Similar to constructors in other OO languages

Typical approach:

- Define an `__init__()` method, with parameters if needed
- Inside the method, set attribute values on the target object
- Perform any additional initialization tasks, if needed

Client code:

- Pass in initialization values when you create an object

Initialization methods (2/2)

Here's an example of how to implement `__init__()`

```
class BankAccount:

    def __init__(self, accountHolder="Anonymous"):
        self.accountHolder = accountHolder
        self.balance = 0.0

    …
```

This is how client code creates objects now

```
acc1 = BankAccount("Fred")
acc2 = BankAccount("Wilma")
```

Making an object's attributes private

One of the goals of OO is encapsulation

- Keep things as private as possible

However, attributes in Python are public by default

- Client code can access the attributes freely!

```
acc1 = BankAccount("Fred")
print("acc1 account holder is %s" % acc1.accountHolder)
```

To make an object's attributes private:

- Prefix the attribute name with two underscores, `__`

```
class BankAccount:

    def __init__(self, accountHolder="Anonymous"):
        self.accountHolder = accountHolder
        self.__balance = 0.0

    …
```

Implementing method behaviour

Here's a more complete implementation of our class

```
class BankAccount:
    """Simple BankAccount class"""

    def __init__(self, accountHolder="Anonymous"):
        self.accountHolder = accountHolder
        self.__balance = 0.0

    def deposit(self, amount):
        self.__balance += amount
        return self.__balance

    def withdraw(self, amount):
        self.__balance -= amount
        return self.__balance

    def toString(self):
        return "{0}, {1}".format(self.accountHolder, self.__balance)
```

Class-wide members

- Class-wide variables
- Class-wide methods
- @classmethod and @staticmethod

Class-wide variables (1/2)

Class-wide variables belong to the class as a whole

- Allocated once, before usage of first object
- Remain allocated regardless of number of objects

To define a class-wide variable:

- Define the variable at global level in the class

```
class BankAccount:
    __nextId = 1
    __OVERDRAFT_LIMIT = -1000
    …
```

To access the class-wide variable in methods:

- Prefix with the class name

Class-wide variables (2/2)

Here's an example that puts it all together

```
class BankAccount:

    __nextId = 1
    __OVERDRAFT_LIMIT = -1000


    def __init__(self, accountHolder="Anonymous"):
        self.accountHolder = accountHolder
        self.__balance = 0.0
        self.id = BankAccount.__nextId
        BankAccount.__nextId += 1


     def withdraw(self, amount):
        newBalance = self.__balance - amount
        if newBalance < BankAccount.__OVERDRAFT_LIMIT:
            print("Insufficient funds to withdraw %f" % amount)
        else:
            self.__balance = newBalance
        return self.__balance

    …
```

Class-wide methods

Typical uses for class-wide methods:

- Get/set class-wide variables
- Factory methods, responsible for creating instances
- Instance management, keeping track of all instances

Example:

```
class BankAccount:

    __nextId = 1
    __OVERDRAFT_LIMIT = -1000
    …

    def getOverdraftLimit():
        return BankAccount.__OVERDRAFT_LIMIT
```

Client code:

```
print("Overdraft limit for all accounts is %d" % BankAccount.getOverdraftLimit())
```

@classmethod and @staticmethod

The @classmethod and @staticmethod decorators can be applied to class-wide methods

```
class BankAccount:

    __OVERDRAFT_LIMIT = -1000
    …

    @classmethod
    def getOverdraftLimit(cls):
        return cls.__OVERDRAFT_LIMIT

    @staticmethod
    def getBanner():
        return "\nThis is the BankAccount Banner"
```

```
# Invoking via the class
print(BankAccount.getBanner())
print(BankAccount.getOverdraftLimit())

# Invoking via an instance
acc1 = BankAccount("Luke")
print(acc1.getBanner())
print(acc1.getOverdraftLimit())
```

Any questions?