import json  
from abc import ABC, abstractmethod  
from types import SimpleNamespace  
  
  
STORAGE = "academy\_storage.json"  
TEACHER\_STORAGE = "teacher\_schedule.json"  
  
  
class Object:  
 *"""transform NameSpace object to json format"""* @staticmethod  
 def toJSON(elem):  
 try:  
 return [x.\_\_dict\_\_ for x in elem]  
 except TypeError:  
 return elem.\_\_dict\_\_  
  
  
class ITeacher(ABC):  
 @abstractmethod  
 def get\_my\_courses(self):  
 pass  
  
 @property  
 @abstractmethod  
 def name(self):  
 pass  
  
 @abstractmethod  
 def \_\_str\_\_(self):  
 pass  
  
  
class Teacher(ITeacher):  
 def \_\_init\_\_(self, name):  
 self.\_\_name = name  
 self.\_\_count\_courses = (  
 len(self.get\_my\_courses()[0]) if len(self.get\_my\_courses()) else 0  
 )  
  
 @property  
 def name(self):  
 return self.\_\_name  
  
 @name.setter  
 def name(self, value):  
 if not isinstance(value, str):  
 raise TypeError("name must be str type")  
 self.\_\_name = value  
  
 def get\_my\_courses(self):  
 with open(TEACHER\_STORAGE, "r") as f:  
 stor = json.load(f)  
  
 return [elem[self.\_\_name] for elem in stor if elem.get(self.\_\_name)]  
  
 def \_\_str\_\_(self):  
 return f"Teacher(name={self.name}, number\_of\_courses={self.\_\_count\_courses})"  
  
  
class ICourse(ABC):  
 @abstractmethod  
 def build\_couse(self):  
 pass  
  
 @abstractmethod  
 def save\_course(self):  
 pass  
  
 @abstractmethod  
 def add\_cours\_to\_teacher\_schedule(self):  
 pass  
  
 @property  
 @abstractmethod  
 def name(self):  
 pass  
  
 @property  
 @abstractmethod  
 def program(self):  
 pass  
  
 @property  
 @abstractmethod  
 def teacher(self):  
 pass  
  
 @abstractmethod  
 def \_\_str\_\_(self):  
 pass  
  
  
class Course(ICourse):  
 def \_\_init\_\_(self, name: str, program: list, teachers):  
 self.name = name  
 self.program = program  
 self.teachers = list(teachers)  
  
 @property  
 def name(self):  
 return self.\_\_name  
  
 @property  
 def teacher(self):  
 return self.\_\_teachers  
  
 @property  
 def program(self):  
 return self.\_\_program  
  
 @name.setter  
 def name(self, value):  
 if not isinstance(value, str):  
 raise TypeError("name must be str type")  
 self.\_\_name = value  
  
 @program.setter  
 def program(self, value):  
 if not isinstance(value, list):  
 raise TypeError("program must be list type")  
 self.\_\_program = value  
  
 @teacher.setter  
 def teacher(self, value):  
 if not (  
 isinstance(value, list) and all(isinstance(elem, Teacher) for elem in value)  
 ):  
 raise TypeError("teacher must be Teacher type")  
 self.\_\_teacher = value  
  
 def build\_couse(self, mode="course") -> dict:  
 *"""create course according to value"""* course = SimpleNamespace()  
 course.name = self.name  
 course.program = self.program  
 if isinstance(self, LocalCourse):  
 course.type = "local"  
 course.laboratory = self.laboratory  
 elif isinstance(self, OffsiteCourse):  
 course.place = self.place  
 course.type = "offsite"  
 if mode == "course":  
 course.teachers = [elem.name for elem in self.teachers]  
 return course  
  
 def save\_course(self):  
 *"""save course into database"""* with open(STORAGE, "r") as f:  
 stor = json.load(f, object\_hook=lambda d: SimpleNamespace(\*\*d))  
  
 if self.build\_couse() not in stor:  
 stor.append(self.build\_couse())  
 with open(STORAGE, "w") as f:  
 json.dump(Object.toJSON(stor), f)  
  
 def add\_cours\_to\_teacher\_schedule(self):  
 *"""add this cource to teacher side in database"""* with open(TEACHER\_STORAGE, "r") as f:  
 stor = json.load(f)  
 a = 0  
 for elem in stor:  
 for teacher in self.teachers:  
 if elem.get(teacher.name):  
 teacher\_schedule = elem[teacher.name]  
 if self.build\_couse("teacher") not in teacher\_schedule:  
 teacher\_schedule.append(  
 Object.toJSON(self.build\_couse("teacher"))  
 )  
 a += 1  
  
 if not a:  
 for teacher in self.teachers:  
 stor.append(  
 {teacher.name: [Object.toJSON(self.build\_couse("teacher"))]}  
 )  
 with open(TEACHER\_STORAGE, "w") as f:  
 json.dump(stor, f)  
  
 def \_\_iter\_\_(self):  
 self.index = 0  
 return self  
  
 def \_\_next\_\_(self):  
 course\_type = "local" if isinstance(self, LocalCourse) else "offsite"  
  
 if self.index < len(CourseFactory.get\_courses(course\_type)):  
 self.index += 1  
 return CourseFactory.get\_courses(course\_type)[self.index - 1]  
 else:  
 raise StopIteration  
  
  
class ILocalCourse(ABC):  
 @abstractmethod  
 def laboratory(self):  
 pass  
  
  
class LocalCourse(Course, ILocalCourse):  
 def \_\_init\_\_(self, name: str, program: list, laboratory: int, teacher: Teacher):  
  
 super().\_\_init\_\_(name, program, teacher)  
 self.laboratory = laboratory  
 self.save\_course()  
 self.add\_cours\_to\_teacher\_schedule()  
  
 @property  
 def laboratory(self):  
 return self.\_\_laboratory  
  
 @laboratory.setter  
 def laboratory(self, value):  
 if not isinstance(value, int):  
 raise TypeError("laboratory must be int type")  
 self.\_\_laboratory = value  
  
 def \_\_str\_\_(self):  
 return f"""LocalCourse(name={self.name}, laboratory={self.laboratory},   
 teachers={[elem.name for elem in self.teachers]})  
 program={self.program}"""  
  
  
class IOffsiteCourse(ABC):  
 @abstractmethod  
 def place(self):  
 pass  
  
  
class OffsiteCourse(Course, IOffsiteCourse):  
 def \_\_init\_\_(self, name: str, program: list, place: str, teacher: Teacher):  
  
 super().\_\_init\_\_(name, program, teacher)  
 self.place = place  
 self.save\_course()  
 self.add\_cours\_to\_teacher\_schedule()  
  
 @property  
 def place(self):  
 return self.\_\_place  
  
 @place.setter  
 def place(self, value):  
 if not isinstance(value, str):  
 raise TypeError("place must be str type")  
 self.\_\_place = value  
  
 def \_\_str\_\_(self):  
 return f"""LocalCourse(name={self.name}, place={self.place},   
 teachers={[elem.name for elem in self.teachers]})  
 program={self.program}"""  
  
  
class ICourseFactory(ABC):  
 @abstractmethod  
 def add\_teacher(self):  
 pass  
  
 @abstractmethod  
 def create\_local\_course(self):  
 pass  
  
 @abstractmethod  
 def create\_offsite\_course(self):  
 pass  
  
 @abstractmethod  
 def get\_all\_courses(self):  
 pass  
  
 @abstractmethod  
 def get\_courses(self, value):  
 pass  
  
  
class CourseFactory:  
 def add\_teacher(self, name):  
 *"""add teacher to database"""* return Teacher(name)  
  
 def create\_local\_course(  
 self, name: str, program: list, laboratory: int, teacher: Teacher  
 ):  
 *"""create local courece"""* return LocalCourse(name, program, laboratory, teacher)  
  
 def create\_offsite\_course(  
 self, name: str, program: list, place: str, teacher: Teacher  
 ):  
 *"""create offsite courece"""* return OffsiteCourse(name, program, place, teacher)  
  
 def get\_all\_courses(self):  
 with open(STORAGE, "r") as f:  
 return json.load(f)  
  
 @staticmethod  
 def get\_courses(course\_type):  
 *"""according to course type return its courses"""* with open(STORAGE, "r") as f:  
 return [elem for elem in json.load(f) if elem["type"] == course\_type]  
  
 def \_\_iter\_\_(self):  
 self.index = 0  
 return self  
  
 def \_\_next\_\_(self):  
 if self.index < len(self.get\_all\_courses()):  
 self.index += 1  
 return self.get\_all\_courses()[self.index - 1]  
 else:  
 raise StopIteration  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 course\_factory = CourseFactory()  
 teacher = course\_factory.add\_teacher("Andriy")  
 teacher2 = course\_factory.add\_teacher("Aboba")  
  
 local = course\_factory.create\_local\_course(  
 "Geography", ["this", "that", "those"], 125, [teacher, teacher2]  
 )  
  
 offsite = course\_factory.create\_offsite\_course(  
 "Geography", ["this", "that", "those"], "Kyiv", [teacher, teacher2]  
 )  
 # for elem in zip(local, offsite):  
 # print(elem)

json

academy\_storage.json

[]

teacher\_schedule.json

[]