AP Computer Science — Basic OOP — Implementing Classes

ORIENTATION

Object-oriented programming (OOP) refers to the organization of data into *classes*, categories that represent a given type of "object." An object can be a model of a real-life thing like a car, a six-sided die, or a person, or it can be a representation of something more abstract like a point in space, a menu item, or a digital clipboard. An object that belongs to a given class is called an *instance* of that class, and any instance of a class keeps track of its *state* using *instance variables*, or *attributes*. The object is created when its *constructor* is called, and can be interacted with by calling its *methods*, which may access data (*accessor methods*, or "getters") or alter the object's data (*mutator methods*, or "setters").

There are two important theoretical concepts in OOP: *abstraction* and *encapsulation*. Objects and the classes they belong to act as *abstractions* of the things they represent: a Java class called `Person` isn't actually a person, of course, but a simplified representation of a person in terms of a limited number of qualities (`name` and `age`, perhaps). Other details such as hair color, nationality, and gender identity aren't included in our abstraction of a person. *Encapsulation* refers to the way that a class and its methods hide away the details of a process. Someone using a `QuadraticFormula` class can call the `hasSolutions()` method to find out whether their equation has solutions without knowing what a discriminant is. The programmer and the class she has written have hidden that information away in a metaphorical "black box."

Basic OOP consists primarily of implementing classes: writing the JavaDocs, class header, constructors, attributes, and methods for an object as specified. Advanced OOP (covered later in this course) consists of designing classes so that they work well together to solve a given problem.

EXAMPLES

Some classes that we've examined that you should be familiar with:

1. **The `Person` class**
   Attributes: `name`, `age`
   Methods: `getName()`, `getAge()`, `changeName(newName)`, `celebrateBirthday()`

2. **The `BankAccount` class**
   Attributes: `balance`
   Methods: `deposit(amount)`, `withdraw(amount)`, `getBalance()`

3. **The `Car` class**
   Attributes: `MPG`, `gas`, `odometer`
   Methods: `addGas(amount)`, `checkGas()`, `getMiles()`, `drive(distance)`

TASKS

Typical things you might be asked to do include:

1. Given a description of a class and its attributes, write the constructor for that class that will initialize instance variables as needed.
2. Given the instance variables for a class, write code that will *declare* those instance variables in the class.
3. Given a description of a method, write the header and body for that method.
4. Identify whether a method is an *accessor* or a *mutator* method.
5. Identify what types of data should be sent in as parameters to a method.
6. Identify what type of data—`int`, `double`, `String`, `void`—should be returned by a method.
7. Explain what the keyword `this` means in the context of instance variables.
8. Given JavaDocs for a class or method, write the class or method.
9. Given a class or method, write the JavaDocs for it.
10. Given a class, write a tester for it.
11. Given a tester, identify what the class might look like.
12. Describe object-oriented programming using key words like classes, objects methods, abstraction, encapsulation, etc.

EXERCISES

1. Write the complete `Car` class. Then write a tester for the `Car` class that demonstrates construction of a `Car` object and the use of the `Car` methods.

2. Modify the `BankAccount` class so that it is a `SavingsAccount` class. Include
   1. an additional instance variable `interest`
   2. an additional constructor that creates a new `SavingsAccount` with a specified `balance` and `interest`
   3. the three methods from the original `BankAccount` class: `deposit()`, `withdraw()`, and `getBalance()`.
   4. an additional mutator method `setInterest`
   5. an additional accessor method `getInterest`
   6. an additional mutator method `addInterest` that increases the `balance` of the account by the `interest` percentage

3. Write the `Dog` class, which describes a dog (and constructs a new dog) in terms of its `name` and its `weight` in pounds (a `double` value). As part of this class:
   1. Write the accessor method `getWeight()`.
   2. Write a mutator method `eatFood(amount)` that increases the dog's weight by the number of pounds specified by the parameter `amount`.
   3. Write a mutator method `poop()` that decreases the dog's weight by 1 or 2 pounds (randomly) each time it is called.
   4. Write a method `speak()` that prints "Bark!" on the screen when called.