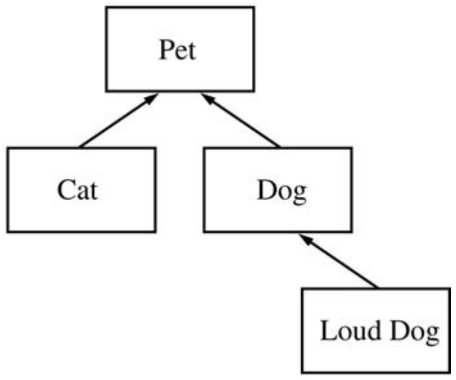
2. Consider the hierarchy of classes shown in the following diagram.



Note that a `Cat` "is-a" `Pet`, a `Dog` "is-a" `Pet`, and a `LoudDog` "is-a" `Dog`.

The class `Pet` is specified as an abstract class as shown in the following declaration. Each `Pet` has a name that is specified when it is constructed.

```
public abstract class Pet
{
  private String myName;

  public Pet(String name)
  {  myName = name;  }

  public String getName()
  {  return myName;  }

  public abstract String speak();
}
```

The subclass `Dog` has the partial class declaration shown below.

```
public class Dog extends Pet
{
  public Dog(String name)
  {  /* implementation not shown */  }

  public String speak()
  {  /* implementation not shown */  }
}
```

GO ON TO THE NEXT PAGE.

(a) Given the class hierarchy shown above, write a complete class declaration for the class `Cat`, including implementations of its constructor and method(s). The `Cat` method `speak` returns `"meow"` when it is invoked.

(b) Assume that class `Dog` has been declared as shown at the beginning of the question. If the `String` *dog-sound* is returned by the `Dog` method `speak`, then the `LoudDog` method `speak` returns a `String` containing *dog-sound* repeated two times.

Given the class hierarchy shown previously, write a complete class declaration for the class `LoudDog`, including implementations of its constructor and method(s).

(c) Consider the following partial declaration of class `Kennel`.

```
public class Kennel
{
  private ArrayList petList;     // all elements are references
                                 // to Pet objects

  // postcondition: for each Pet in the kennel, its name followed
  //                by the result of a call to its speak method
  //                has been printed, one line per Pet
  public void allSpeak()
  {  /* to be implemented in this part */  }

  // ... constructor and other methods not shown
}
```

Write the `Kennel` method `allSpeak`. For each `Pet` in the kennel, `allSpeak` prints a line with the name of the `Pet` followed by the result of a call to its `speak` method.

In writing `allSpeak`, you may use any of the methods defined for any of the classes specified for this problem. Assume that these methods work as specified, regardless of what you wrote in parts (a) and (b). Solutions that reimplement functionality provided by these methods, rather than invoking these methods, will not receive full credit.

Complete method `allSpeak` below.

```
  // postcondition: for each Pet in the kennel, its name followed
  //                by the result of a call to its speak method
  //                has been printed, one line per Pet
  public void allSpeak()
```

GO ON TO THE NEXT PAGE.

3. This question involves reasoning about the code from the Marine Biology Simulation case study. A copy of the code is provided as part of this exam.

Consider adding a `PondStocker` class to the Marine Biology Simulation case study. The purpose of the `PondStocker` class is to increase the fish population in a pond if the population density falls below a certain minimum. The population density of an environment is the ratio of the number of fish in the environment to the total number of locations in the environment. When there are too few fish in the environment, enough fish will be added to make the population greater than the specified minimum density. You will be asked to implement some of the methods for the `PondStocker` class. The declaration of the `PondStocker` class is as follows.

```
public class PondStocker
{
  private Environment theEnv;
  private double minDensity;    // 0.0 <= minDensity < 1.0

  // postcondition: returns the minimum number of fish that need to be
  //                added to make the population density greater than
  //                minDensity
  private int numUnder()
  {  /* to be implemented in part (a) */  }

  // postcondition: returns a random location within the bounds of theEnv
  private Location randomLocation()
  {  /* to be implemented in part (b) */  }

  // precondition:  0 <= numToAdd <= number of empty locations in theEnv
  // postcondition: the number of fish in theEnv has been increased
  //                by numToAdd; the fish added are placed at
  //                random empty locations in theEnv
  public void addFish(int numToAdd)
  {  /* to be implemented in part (c) */  }

// constructor and other methods not shown
}
```

For example, suppose that the environment has 7 rows and 7 columns, giving it a total of 49 cells. If the minimum density is 0.5, 25 cells need to be occupied to meet the minimum density requirement. If the number of fish in the environment is 17, then the call `numUnder()` would return 8.

GO ON TO THE NEXT PAGE.

AP® Computer Science A
2004 SCORING GUIDELINES

Question 2

Part A:	`class Cat`	**2 pts**

+1/2 `public class Cat extends Pet`

+1/2 Constructor correct (must call `super`)

+1 `speak` method
- **+1/2** attempt (method header matches abstract method, OK if `abstract` left in)
- **+1/2** correct

Part B:	`class LoudDog`	**3 pts**

+1/2 `public class LoudDog extends Dog`

+1 Constructor correct (must call `super`)

+1 1/2 `speak` method
- **+1** attempt (calls `super.speak()` *and* method header matches abstract method, OK if `abstract` left in)
- **+1/2** correct value returned

Part C:	Kennel - `allSpeak`	**4 pts**

+1 loop over `petList`
- **+1/2** attempt
- **+1/2** correct (must access `petList`)

+1 1/2 get pet from `petList` (no deduction for missing downcast from `petList`)
- **+1/2** attempt
- **+1** correct (local variable must be type `Pet`)

+1 1/2 print `p.getName()` and `p.speak()` for pet `p` (local variable not necessary)
- **+1/2** attempt (must have `xxx.getName()` or `xxx.speak()`, for some xxx)
- **+1** correct

Note: if done in-line with no local, no deduction for missing downcast.

Usage:
- **-1/2** `public` instance variable
- **-1** parent class name instead of `super`
- **-1/2** `getName` is overridden (other than `super.getName`) in part (a) and/or part (b)

(No deduction for other additional methods or constructors.)