420-200-VA | Assignment 4-A | Due Date: 12 April 2020

Inheritance and polymorphism

1 Purpose

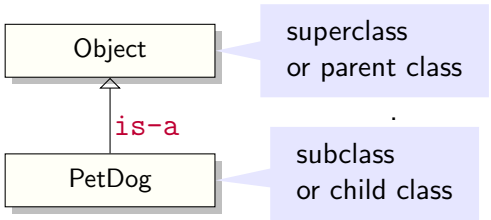
To practice OOP using inheritance, polymorphism, `abstract` classes and `abstract` methods

2 Java Inheritance

One of the key principles of Object-Oriented Programming (OOP) is inheritance, which allows us to *extend* an existing class to create a new one.

2.1 Class Object at the Top of every Java Inheritance Hierarchy

In Java, every class we use or write `extends` the `Object`[1] class, directly or indirectly. Our `PetDog` class is no exception. The inheritance hierarchy shown in the UML[2] class diagram at right represents an `is-a` relationship between `PetDog` and `Object` classes, indicating that `PetDog` inherits the features of `Object`.

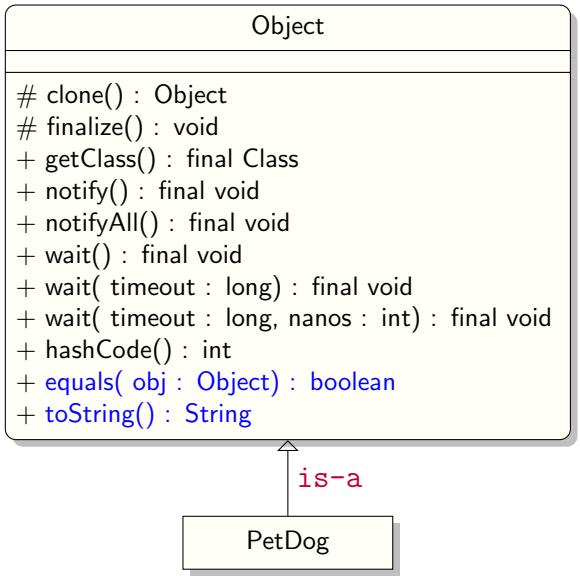


The features of a class are its properties (data members) and its functionality (methods). To see the features of class `Object` that are actually inherited by `PetDog`, we expand class `Object` into its members as shown below. Recall that the symbols +, #, − represent, respectively, the `public`, `protected`, and `private` members of a class in UML.

As you can see, although it doesn't reveal its own features, class `PetDog` explicitly reveals all the 11 methods it inherits from class `Object`, and since none of those methods is `private`, any method in `PetDog` can call any of the 11 methods in class `Object` as if they were all its own.

Note that just because a subclass inherits all members of its superclass, it doesn't mean that the subclass can "access" every method and and every field in the superclass. Specifically, a subclass *cannot* access the private members of the superclass.

A subclass method can both overload and override an instance method of the superclass. As you know, methods equals and toString are the most frequently overridden methods by other Java classes.



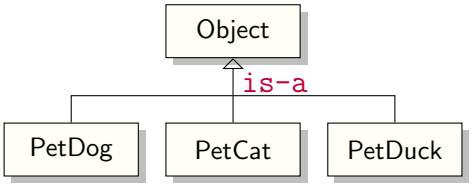
[1] See Nathan Schutz's video at 4:05 here

[2] We covered UML notation in class and in assignment 1. It is reproduced at the end of this document FYI.

Since the ever present `Object` class will always be at the top of every inheritance hierarchy in Java, the remaining UML class diagrams in this assignment will not depict it expanded.

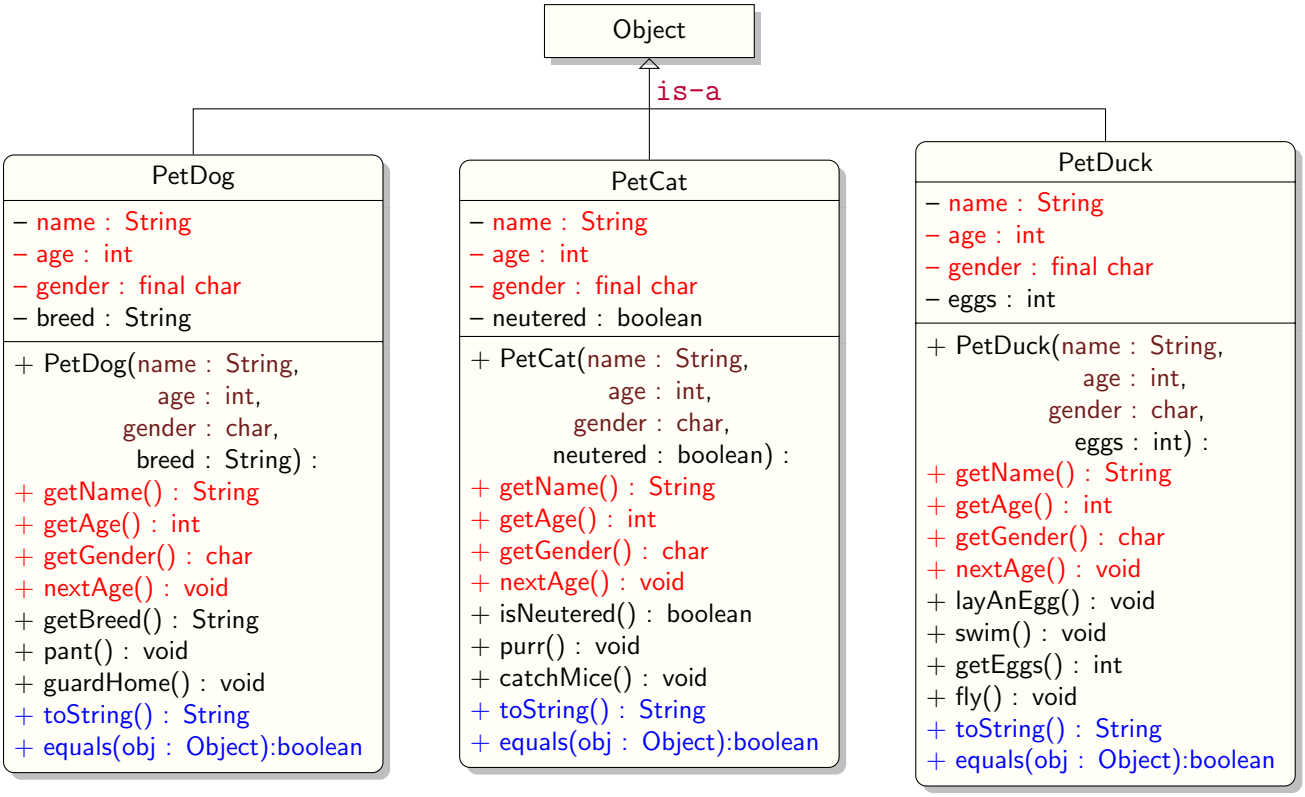
2.2 Code Reuse

Representing an `is-a` relationship between classes, inheritance allows a superclass to share its non-`private` features with its subclasses.



For example, consider classes `PetDog`, `PetCat`, and `PetDuck` depicted in the UML inheritance diagram at right:

In addition to the features they already inherent from their superclass `Object`, classes `PetDog`, `PetCat`, and `PetDuck` each have their own features. For example, a dog pants and guards home; a cat catches mice and purrs; and a duck swims, flies, and lays eggs. Specifically:



The methods equals and toString each represent an override of same methods with exact same signatures in superclass `Object`.

The code in red represents the common features of classes `PetDog`, `PetCat`, and `PetDuck`, highlighting *code duplication* in multiple places, which results in a huge code maintenance problem.[3] For starters, just take a look at obvious code duplication in the constructors:

[3] Imagine having to edit only the red code in three or thirty or even more pet classes in exactly the same way, leaving anything else in each file intact and error free!

```java
public PetDog(String name, int age, char gender, String breed)
{
    this.name = name;
    this..age = age;
    this.gender = gender;
   this.breed = breed;
}
```

```java
public PetCat(String name, int age, char gender, boolean neutered)
{
    this.name = name;
    this..age = age;
    this.gender = gender;
   this.neutered = neutered;
}
```

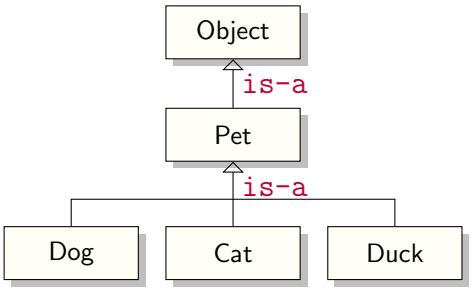
```java
public PetDuck(String name, int age, char gender, int eggs)
{
    this.name = name;
    this..age = age;
    this.gender = gender;
   this.eggs = eggs;
}
```

As you can see, similar code duplication exists in all of the getter and setter methods, `toString` and `equals`, etc.

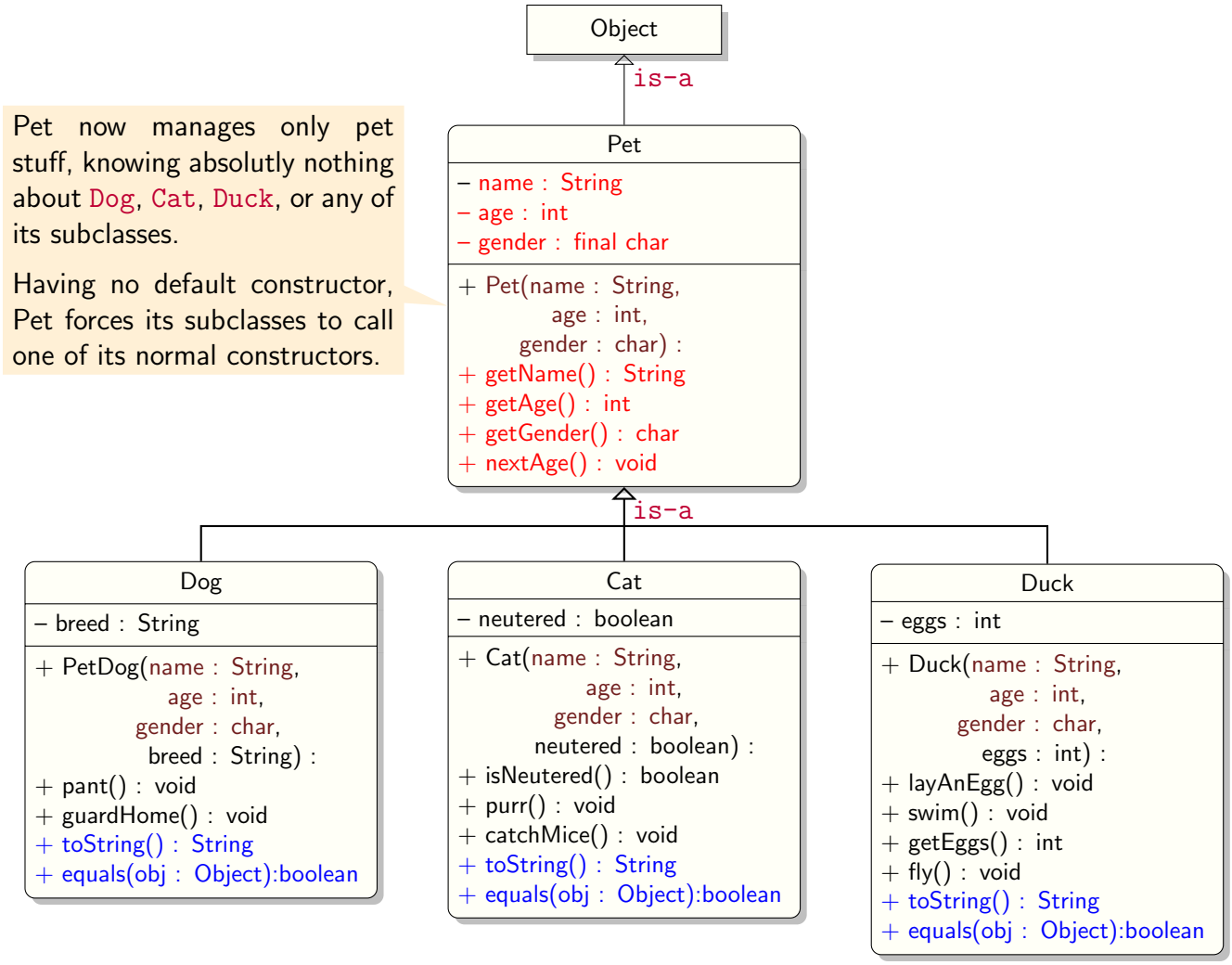
2.3 Code Refactoring

Code refactoring is a technique for restructuring and simplifying existing code. Our goal here is to eliminate code duplication by consolidating the common features of `PetDog`, `PetCat`, and `PetDuck` into a shared superclass `Pet`.

This refactoring process will reduce `PetDog`, `PetCat`, and `PetDuck` to three lean subclasses of `Pet`, which we simply name `Dog`, `Cat`, and `Duck`.



Expanding the classes involved (except for class `Object`) will result in the following UML class diagram:



2.4 No Child Object Can Exist without a Parent Object

The construction of an object of subclasses Dog, Cat, or Duck cannot begin until after the construction of superclass Pet has been completed. Here is how objects of Pet are constructed:

```
class Pet
{  String name;
   int age;
   char gender;
   // initializes this pet object; one piece of code regardless of number of subclasses
   public Pet(String name, int age, char gender)
   {
       this.name = name;
       this..age = age;
       this.gender = gender;
   }
   // other code not shown for brevity   // ...
}
```

Here is how objects of Pet subclasses Dog, Cat, and Duck are constructed:

```
class Dog extends Pet
{
   String breed;
   // initializes this dog object
   public Dog(String name, int age, char gender, String breed)
   {  // a call to a super's constructor is mandatory here;
      // must be the very first non-comment statement;
      // first, have a super's constructor initialize Pet properties
       super(name, age, gender);
     // next, have this subclass constructor initialize Dog properties
      this.breed = breed;
   }
   // other code not shown for brevity   // ...
}
```

```
class Cat  extends Pet
{
   boolean neutered;
   // initializes this cat object
   public Cat(String name, int age, char gender,  boolean neutered)
   {  // a call to a super's constructor is mandatory here;
      // must be the very first non-comment statement;
      // first, have a super's constructor initialize Pet properties
       super(name, age, gender);
     // next, have this subclass constructor initialize Cat properties
      this.neutered = neutered;
   }
   // other code not shown for brevity   // ...
}
```

```
class Duck  extends Pet
{
   int eggs;
   // initializes this duck object
   public Duck(String name, int age, char gender, int eggs)
   {  // a call to a super's constructor is mandatory here;
      // must be the very first non-comment statement;
      // first, have a super's constructor initialize Pet properties
       super(name, age, gender);
     // next, have this subclass constructor initialize Duck properties
      this.eggs = eggs;
   }
   // other code not shown for brevity   // ...
}
```

2.5 Let Super's toString() do its thing

When overriding the `toString()` method in a subclass, delegate (pass the buck) to `super`'s `toString()` to do its thing, as opposed to repeating `super`'s `toString()` in the subclass:

```
class Pet
{
   @Override public String toString()
   { char strGender = Character.toUpperCase(gender);
      return "I'm " + name + ", a " + age + " year old "
              + ((strGender == 'M') ? "male" : "female") + " pet";
   }
   // other members not shown for brevity
}
```

```
class Dog extends Pet
{
   @Override public String toString()
   {
      String result = super.toString(); // let super do its toString() thing
      result += " "  + breed + " dog";  // let subclass do its toString() thing
      return result;
   }
   // other members not shown for brevity
}
```

```
class Cat  extends Pet
{
   @Override public String toString()
   {
      String result = super.toString(); // let super do its toString() thing
      result += " "  + (neutered?"":"not ") + "neutered cat"; // let subclass do its toString()
      return result;
   }
   // other members not shown for brevity
}
```

```
class Duck  extends Pet
{
   @Override public String toString()
   {
      String result = super.toString(); // let super do its toString() thing
      result += " duck with "  + eggs + " eggs"; // let subclass do its toString() thing
      return result;
   }
   // other members not shown for brevity
}
```

2.6 Let Super's equals() do its thing

When overriding the `equals()` method in a subclass, delegate (pass the buck) to `super`'s `equals()` to do its thing, as opposed to repeating `super`'s `equals()` thing in the subclass:

```
class Pet
{
   @Override
   public boolean equals(Object obj)
   {
      if (this == obj) return true;
      if (obj == null) return false;
      if (getClass() != obj.getClass()) return false;

      final Pet other = (Pet) obj;
      if (this.age != other.age) return false;
      if (this.gender != other.gender)  return false;
      if (!(this.name.equals(other.name))) return false;
      return true;
   }
   // other members not shown for brevity
}
```

```
class Dog extends Pet
{
   @Override
   public boolean equals(Object obj)
   {
      if (this == obj) return true;
      if (obj == null) return false;
      if (getClass() != obj.getClass()) return false;

      final Dog other = (Dog) obj;
      boolean result = super.equals(obj); // let super do its equals() thing
      result = result && this.breed.equals(other.breed); // let subclass do its equals() thing
      return result;
   }
   // other members not shown for brevity
}
```

```
class Cat  extends Pet
{
   @Override
   public boolean equals(Object obj)
   {
      if (this == obj) return true;
      if (obj == null) return false;
```

```
        if (getClass() != obj.getClass()) return false;

        final Cat other = (Cat) obj;
        boolean result = super.equals(obj); // let super do its equals() thing
        result = result && neutered == other.neutered; // let subclass do its equals() thing
        return result;
    }
    // other members not shown for brevity
```

```
class Duck  extends Pet
{
    @Override
    public boolean equals(Object obj)
    {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;

        final Duck other = (Duck) obj;
        boolean result = super.equals(obj); // let super do its equals() thing
        result = result && eggs == other.eggs; // let subclass do its equals() thing
        return result;
    }
    // other members not shown for brevity
}
```

3 Task 1 of 2

Implement the Pet, Dog, Cat, and Duck classes above.

You may have noticed that none of the class diagrams specifies methods to set the values of the age and gender fields. That's because after initialization, gender remains fixed, and age can be changed only by the nextAge() method, which increments age by 1.

Your implementation must support the following driver code:

```
public class Assignment_4A_Driver
{
    public static void main(String[] args) {
        Dog fido = new Dog("Daisy", 5, 'f', "pug");
        System.out.println(fido);
        fido.nextAge(); System.out.println(fido);
        fido.nextAge(); System.out.println(fido);
        fido.nextAge(); System.out.println(fido);
        fido.guardHome();
        fido.pant();

```

```
        boolean spayed = true;
        Cat garfield = new Cat("Garfield", 5, 'm', spayed);
        System.out.println(garfield);
        garfield.nextAge(); System.out.println(garfield);
        garfield.catchMice();
        garfield.purr();

        Duck daffy = new Duck("Daffy", 4, 'F', 5);
        System.out.println(daffy);
        daffy.nextAge();  System.out.println(daffy);
        daffy.nextAge();  System.out.println(daffy);
        daffy.layAnEgg(); System.out.println(daffy);
        daffy.layAnEgg(); System.out.println(daffy);
        daffy.swim();
        daffy.fly();

        // Practice Polymorphism
        // code not shown
        return;
    }
}
```

Output

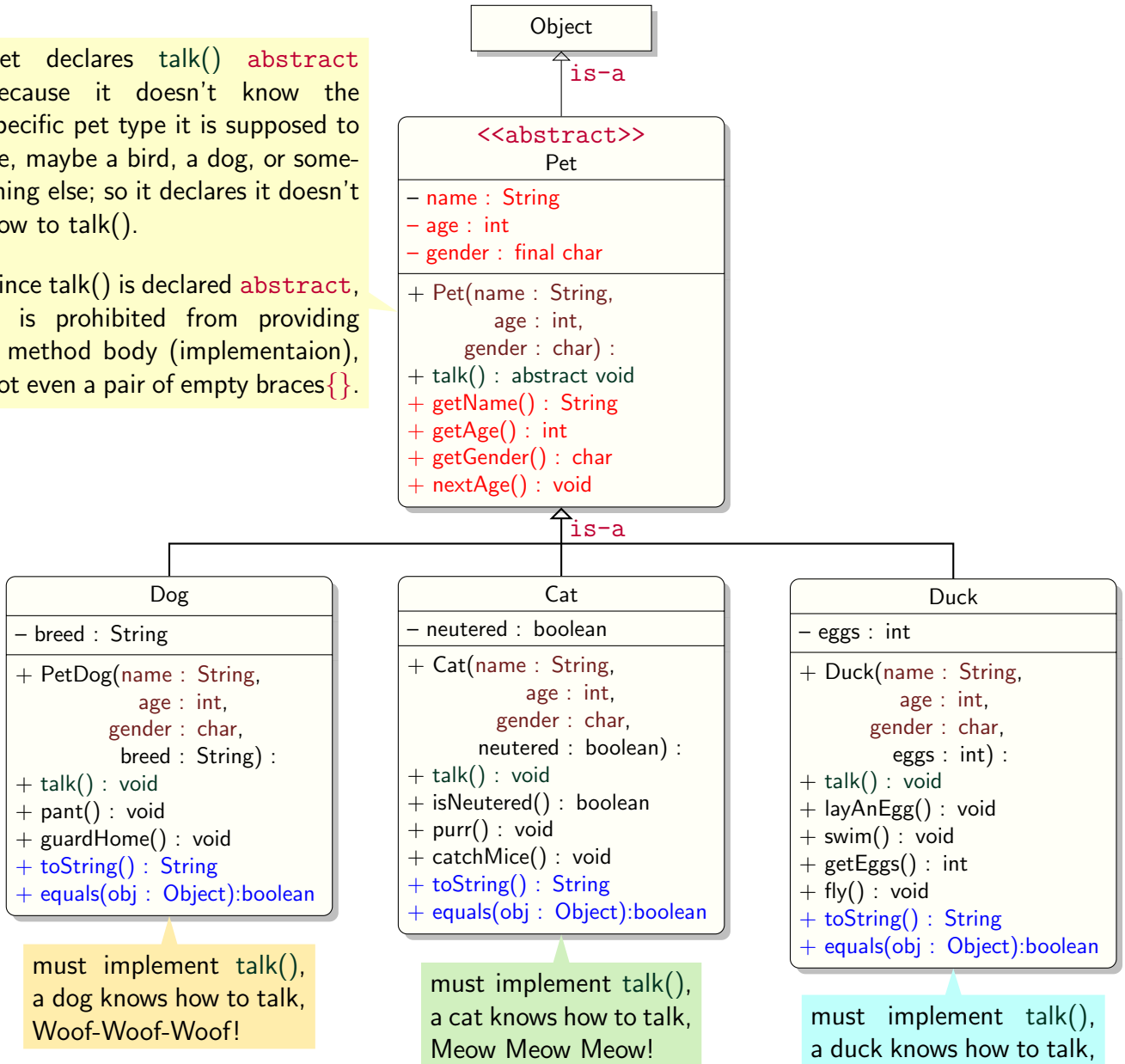
```
I'm Daisy, a 5 year old female pet pug dog
I'm Daisy, a 6 year old female pet pug dog
I'm Daisy, a 7 year old female pet pug dog
I'm Daisy, a 8 year old female pet pug dog
Daisy is guarding home ...
Daisy is panting ...
I'm Garfield, a 5 year old male pet neutered cat
I'm Garfield, a 6 year old male pet neutered cat
Garfield is catching mice ...
Garfield is purring ...
I'm Daffy, a 4 year old female pet duck with 5 eggs
I'm Daffy, a 5 year old female pet duck with 5 eggs
I'm Daffy, a 6 year old female pet duck with 5 eggs
Daffy just laid  an egg ...
I'm Daffy, a 6 year old female pet duck with 6 eggs
Daffy just laid  an egg ...
I'm Daffy, a 6 year old female pet duck with 7 eggs
Daffy is  swimming ...
Daffy is  flying ...
```

4 Polymorphism: Modeling Talking Pets

Enhance your implementation of the `Pet` class to include an `abstract void talk();` method, then *override* the `talk()` method in the `Dog`, `Cat`, and `Duck` subclasses. Remember that the presence of at least one `abstract` method in a class turns that class into an `abstract class`.

Pet declares talk() abstract because it doesn't know the specific pet type it is supposed to be, maybe a bird, a dog, or something else; so it declares it doesn't how to talk().

Since talk() is declared abstract, it is prohibited from providing a method body (implementaion), not even a pair of empty braces{}.



must implement talk(), a dog knows how to talk, Woof-Woof-Woof!

must implement talk(), a cat knows how to talk, Meow Meow Meow!

must implement talk(), a duck knows how to talk, Quack-Quack-Quack!

5 Task 2 of 2

Enhance your implementation of the `Pet` class as described above.

Your implementation must support the following driver code:

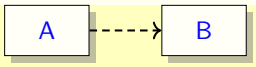
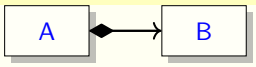
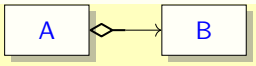
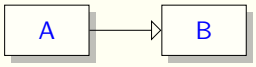
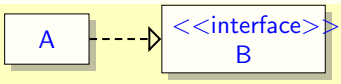
```
public class Assignment_4A_Driver
{
   public static void main(String[] args) {
        Dog fido = new Dog("Daisy", 5, 'f', "pug");
        System.out.println(fido);
        fido.nextAge(); System.out.println(fido);
        fido.nextAge(); System.out.println(fido);
        fido.nextAge(); System.out.println(fido);
        fido.guardHome();
        fido.pant();

        boolean spayed = true;
        Cat garfield = new Cat("Garfield", 5, 'm', spayed);
        System.out.println(garfield);
        garfield.nextAge(); System.out.println(garfield);
        garfield.catchMice();
        garfield.purr();

        Duck daffy = new Duck("Daffy", 4, 'F', 5);
        System.out.println(daffy);
        daffy.nextAge();  System.out.println(daffy);
        daffy.nextAge();  System.out.println(daffy);
        daffy.layAnEgg(); System.out.println(daffy);
        daffy.layAnEgg(); System.out.println(daffy);
        daffy.swim();
        daffy.fly();

        // Practice Polymorphism
        Pet p; // declare a variable of the superclass;
        // now, let's demonstrate an axample of polymorphism;
        // let p reference an object of Dog, Cat, or Duck at random
        int roll = (new Random()).nextInt(3); // roll is 0, 1, or 2, but which one?
        if ( roll == 0)
        {  // a Pet variable referencing a Dog obj
           p = new Dog("Oscar", 10, 'M', "Great Dane");
        }
        else if ( roll == 1)
        {  // a Pet variable referencing a Cat obj
           p = new Cat("Sassy", 7, 'F', false);
        }
        else //  roll is 2
        {   // a Pet variable referencing a Duck obj
           p = new Duck("Donald Duck", 3, 'm', 5);
        }

```

```
46          // here, our variable p is referencing a Dog, Cat, or Duck object;
47          // which object is the Pet variable p referencing?
48          // we don't exactly know which one, but the runtime system does!
49          System.out.println("calling the talk() method polymorphically");
50          p.talk();
51          System.out.println(p);
52  // here is polymorphism in a nutshell:
53  // p is a variable of Pet, our super class <-- very very very important!
54  // P can reference an object of any subclass of Pet;
55  // at runtime, if p references a Dog object, then Dog's talk() is called
56  // at runtime, if p references a Cat object, then Cat's talk() is called
57  // at runtime, if p references a Duck object, then Duck's talk() is called
58  // run this program again and again to see polymorphism in action!
59  // pay attention to lines 21-22 on the output
60          return;
61      }
62  }
```

Output

```
1  I'm Daisy, a 5 year old female pet pug dog
2  I'm Daisy, a 6 year old female pet pug dog
3  I'm Daisy, a 7 year old female pet pug dog
4  I'm Daisy, a 8 year old female pet pug dog
5  Daisy is guarding home ...
6  Daisy is panting ...
7  I'm Garfield, a 5 year old male pet neutered cat
8  I'm Garfield, a 6 year old male pet neutered cat
9  Garfield is catching mice ...
10 Garfield is purring ...
11 I'm Daffy, a 4 year old female pet duck with 5 eggs
12 I'm Daffy, a 5 year old female pet duck with 5 eggs
13 I'm Daffy, a 6 year old female pet duck with 5 eggs
14 Daffy just laid  an egg ...
15 I'm Daffy, a 6 year old female pet duck with 6 eggs
16 Daffy just laid  an egg ...
17 I'm Daffy, a 6 year old female pet duck with 7 eggs
18 Daffy is  swimming ...
19 Daffy is  flying ...
20 calling the talk() method polymorphically
21 Woof-Woof-Woof!
22 I'm Oscar, a 10 year old male pet Great Dane dog
```

6 Summay of UML Class Diagram Notation

Relationship	UML notation	Meaning
Dependency		An `A` uses (depends on) a `B`. Basically, an `A`'s method has a `B` reference, as a local variable, a parameter, or both.
Composition		An `A` is made up of `B`s with lifetime dependency. The `B`s belong to `A`; if `A` is destroyed, its `B`s are destroyed as well.
Aggregation		An `A` is made up of `B`s. The `B`s are shared with `A`.
Inheritance		An `A` *is a* special kind of `B`. `A` specializes `B`. `B` generalizes `A`. `A` has access to all non-private members of `B`.
Interface		An `A` uses (requires) the interface `B`. The `A` class must implement the operations specified in the B interface.

7 What Members Can be Accessed from Which Classes?

Members with this access type	Can be accessed from			
	the same class	another class in the same package	a subclass	another class in the outside world
`private`	✓			
	✓	✓		
`protected`	✓	✓	✓	
`public`	✓	✓	✓	✓

By not writing any access modifier at all before a method or a field of a class, we give that member the default (package) access.

Such members are accessible from any method in any class in the same package, which is effectively a folder.

8 Evaluation Criteria

Evaluation Criteria

Functionality	Ability to perform as required, producing correct output for any set of input data, Proper implementation of all specified requirements, Efficiency	60%
Robustness	Ability to handle input data of wrong type or invalid value	10%
OOP style	Encapsulating only the necessary data inside objects, Information hiding, Proper use of Java constructs and facilities.	10%
Documentation	Description of purpose of program, Javadoc comment style for all methods and fields, comments on non-trivial steps in all methods	10%
Presentation	Format, clarity, completeness of output, user friendly interface	5%
Code readability	Meaningful identifiers, indentation, spacing, localizing variables	5%