Chapter 7 – Classes & Objects, Part B

These note present *Dog* simulation example that shows how we go about OO modeling. A number of new things are introduced. They also present the *Person –> BirthDate* example.

Object Oriented Introduction

Next, we will see an example object-oriented (OO) modeling. Our objective is to begin to learn to *think OO*.

1. We will write a simple computer *simulation.* We want to *simulate* taking a dog for a walk and we are interested in counting how many times the dog stops for a drink of water (assume water is always available a very short distance away).

   To do a simulation, we must make some assumptions and gather some data. So, we will walk the dog 2 miles. and we will break up the walk into a bunch of very short walks, each one a different length, a random length. At the conclusion of each short walk, we will see if the dog is thirsty and if so, let it drink, and then take another short walk. So, the algorithm looks like this:

   ```
   Loop while total distance walked < 2 miles
       Dog walks short (random) distance
       If dog is thirsty
           Dog drinks
   ```

   Consider a more complete algorithm and the corresponding code:

   OO Algorithm (for *main()*)

   ```
   Create a dog
   While dog has walked less than 2 miles
       Dog walks
       If dog is thirsty
           Dog drinks
           Increment number of drinks
   ```

   Code

   ```
   Dog d = new Dog("Max");

   while( d.distanceWalked < 2 )

       d.walk();

       if( d.isThirsty )

           d.drink();

   System.out.println( d.numDrinks );
   ```

   Next, consider a *procedural* algorithm:

   Procedural Pseudo-Code (for *main()*)

   ```
   While totalDistance less than 2 miles
       d = getDistanceWalked()
       totalDistance = d;
       lastWater += d
       If isThirsty()
           Increment number of drinks
   ```

   Code

   ```
   double dist=0.0;

   while( dist < 2 )

       dist += getDistanceWalked();

       if( isThirsty() )
           numDrinks++;
   ```

What is the difference between the *procedural* pseudo-code shown above and the OO code? Both have methods that do similar types of things. But, the difference is that the OO code attributes those methods to an *object:* a *dog drinks* and a *dog walks*. Notice, that using the OO approach, we have described the algorithm in a *natural* way, as *objects* that have behaviors and interact with their surrounding world. This is the *object-oriented (OO)* approach to modeling. It is now our goal to:

a. Learn to *think* in OO. This is a real challenge and it only comes with experience. The more systems (programs) you model (write), the better you will get at it. It is a lifelong learning process. To begin this process, of learning to model a problem (system) using OO, we ask ourselves these questions:

   i. What *objects* are naturally present in problem/system? They are almost always nouns. For instance: *person*, *dog, gradeBook*, *building*, *game*, *temperature*. However, not all nouns are objects. They may simply be objects we don't need to represent for the current problem. Or, they may be *responsibilities...*

   ii. What are these objects' *responsibilities*? What things should the object be able to do? Keep track of? What *behaviors* does it have? For instance, we may want a *gradeBook* to be able to tell use the highest score on a particular test, or provide all the scores for a test; we may want a *building* to be able to provide us with a list of rooms, a list of names of people who can access a room; a *temperature* object may need to update itself.

b. Learn the mechanics for writing classes, creating and using objects. Mostly, the focus will be on this.

2. A *class* is a template for making *objects*. We use the *Dog* class to make a *dog* object. A class has *properties* and *methods*.

A *property* (also called *attribute* or *field*) is a feature objects created from this class have, they may hold information about the *state* of an object. For instance, in the example above, we say that *distanceWalked, numDrinks,* and *isThirsty* are *properties* of the *Dog* class. You use a property like you use a method, except that properties don't have parameters and thus don't end with ().

```
dist = d.distanceWalked;
boolean isThr = d.isThirsty;
int nd = d.numDrinks;
```

A *method* is a *responsibility* (*behavior*) associated with the class. *walk()* and *drink()* are methods of the *Dog* class.

```
d.walk();
d.drink();
```

Methods and properties implement the *responsibilities* of the class.

There are two aspects to OO programming:

a. Writing the Java code that defines *classes* (writing *classes*) that are needed to solve the problem.

b. Writing programs that use *objects* created from the *classes* to solve the problem.

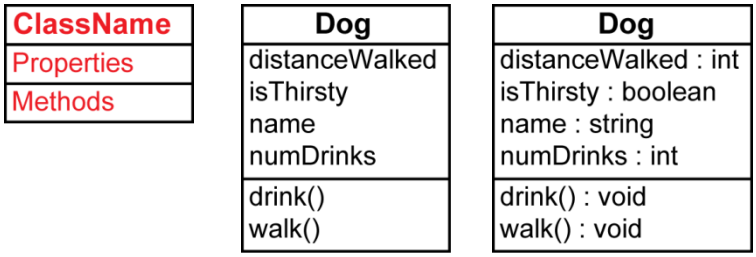
3. The dog, *d* is an *object* and *Dog* is a *class*.

```
Dog d = new Dog("Max");
```

We say that *d* is

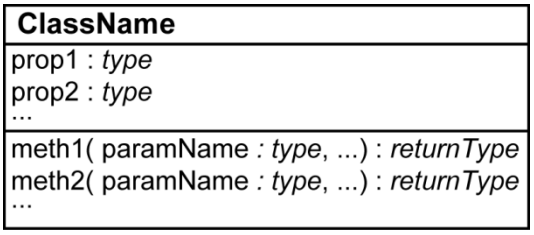
a. an object created from the *class Dog*.
b. an object of *type Dog*
c. a *Dog* object
d. an instance of a *Dog*
e. a *Dog*

4. A convenient way to represent a class is to use UML (Unified Modeling Language). We model a class with a figure as shown in (a). For instance the Dog class can be represented as in (b) or (c), where the data types and return types are also shown. We call either (b) or (c) a class diagram.



(a) (b) (c)

5. For now, the syntax for a class diagram looks like this:



Writing a Class

1. Now, we need some details to actually do the simulation referred to earlier. We will assume that each *short walk* is a random distance between 25 and 75 yards. We will assume that at the conclusion of each *short walk*, the dog can *get thirsty* in two different ways. The first is that the dog just randomly gets thirsty which occurs with probability 0.33. Second, if the total distance since the last drink is more than 300 yards then it automatically will be thirsty.

2. We need to write the *Dog* class, but before we can do it, we must understand what properties and methods it needs. To adopt an OO mindset, we need to ask, "what should a dog be keeping up with in the context of this problem?" Probably, we see that he needs to keep up with how far he has walked, and if he is thirsty or not, and his name. We also ask, "what should a dog be able to do in the context of this problem?" Again, probably we see that the dog needs to be able to walk and to drink. And we see that when the dog walks, this affects the total distance walked. The code for the Dog class:

```
class Dog
{
        private final int MAX_THRIST_LEVEL = 300;
        private final double HUNGER_PROBABILITY = 0.33;
        private int distSinceLastWater;

        public Dog( String dogName )
        {
              distSinceLastWater = 0;
              isThirsty = false;
              name = dogName;
        }

        public String name;
        public boolean isThirsty;
        public int distanceWalked;
        public int numDrinks = 0;

        public void walk()
        {
              int dist = 25 + (int)(Math.random()*50 + 1 );

              distanceWalked += dist;
              distSinceLastWater += dist;

              if ( ( Math.random() < HUNGER_PROBABILITY ) ||
                   ( distSinceLastWater > MAX_THRIST_LEVEL  ) )

                    isThirsty = true;
        }
        public void drink()
        {
              numDrinks++;
              isThirsty = false;
              distSinceLastWater = 0;
        }
}
```

3. We notice that there is one extra "method" in the Dog class:

```
public Dog( String dogName )
{
      distSinceLastWater = 0;
      isThirsty = false;
      name = dogName;
}
```

This is called the *class constructor* (or just *constructor*). Its job is to create a new object (dog) from the class. For instance, in the main() (or in some other method), when we write:

```
Dog d = new Dog("Max");
```

the constructor is automatically called and creates a Dog object which we *reference* with the variable *d*. Notice, also that the constructor takes an argument, *dogName*. It assigns this value, "Max", to the *name property*. This, in essence, *saves* the dog's name inside the dog object.

4. Notice that the Dog class has some *private* variables (private instance variables). They are variables that are used *inside* a dog *object* but cannot be referred to from *outside* the dog object. For instance, in the main(), we **cannot** write this code:

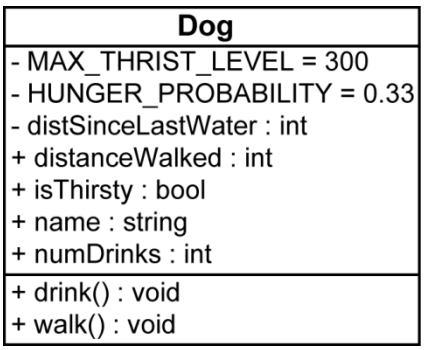
```
d.distSinceLastWater
```

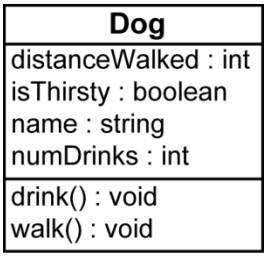
5. The *public* variables in the Dog class are referred to as *properties* of the Dog class (sometimes called *attributes*). The fact that they are public means that they *are* accessible outside a dog object. For instance, in the main(), we **can** write this code:

```
d.isThirsty
```

6. Similarly, the public methods in the Dog class are called the *behaviors* of a dog object and can be referenced outside a dog object. Frequently, we also have private methods, which can be used inside an object, but not outside. These private methods are sometimes referred to as *helpers*, they help the public methods accomplish their tasks. We will see examples of these shortly.

7. A more complete class diagram for the Dog class is shown on the below on the left. However, many times we are more interested in the *public interface* of the class as shown below in the right two diagrams. Notice that we haven't shown the constructor below; however, it is OK to do so.





Using a Class

1. Next, we will add a *main()* to the class so that we can run our program:

```
public static void main(String[] args)
{
      final int MAX_WALK_DISTANCE = 2*1260;

      Dog d = new Dog("Max");

      while( d.distanceWalked < MAX_WALK_DISTANCE )
      {
            d.walk();

            if( d.isThirsty )
            {
                  d.drink();
            }
      }

      double numMiles = (double)d.distanceWalked / 1260;

      String strNumMiles = String.format(
                              "Dog: '%s' walked: %.3f miles" +
                              " and stopped to drink water %d times.",
                              d.name, numMiles, d.numDrinks );

      System.out.println( strNumMiles );
}
```

Modifying the Problem

1. Now, we consider a modification to the program: We want to simulate two dogs instead of one. Thus, the *main()* will look like this:

```
final int MAX_WALK_DISTANCE = 2*1260;

Dog d1 = new Dog("Max");
Dog d2 = new Dog("Buster");

while( ( d1.distanceWalked < MAX_WALK_DISTANCE ) ||
       ( d2.distanceWalked < MAX_WALK_DISTANCE ) )
{
      d1.walk();
      d2.walk();

      if( d1.isThirsty ) d1.drink();
      if( d2.isThirsty ) d2.drink();
}

double numMiles = (double)d1.distanceWalked / 1260;

String strMsgDog1 = String.format( "Dog: '%s' walked: %.3f miles" +
                    " and stopped to drink water %d times.",
                      d1.name, numMiles, d1.numDrinks );

System.out.println( strMsgDog1 );

numMiles = (double)d2.distanceWalked / 1260;

String strMsgDog2 = String.format( "Dog: '%s' walked: %.3f miles" +
                    " and stopped to drink water %d times.",
                      d2.name, numMiles, d2.numDrinks );

System.out.println( strMsgDog2 );
```

A Pack of Dogs

1. Let's make another modification. We will now consider walking a *pack* of dogs. We'll use an *array of Dogs* to represent the *pack*.

```
Dog[] pack = new Dog[ NUM_DOGS ];
```

Arrays work similar to the way they do for primitive data types, except each element *references* an *object* from a particular class. With primitive data types, the elements themselves contain the values. Again, we'll be lazy and say, "array of dogs," when we really should say, "array of references to dogs." Here is the *main()* for the example:

2. Code:

```
int count;
final int MAX_WALK_DISTANCE = 2*1260;
double numMiles = 0.0;

final int NUM_DOGS = 5;
boolean dogsFinished = false;

// Create pack of dogs array.
Dog[] pack = new Dog[ NUM_DOGS ];

// Create dogs in pack.
for( int i=0; i<pack.length; i++ )
   pack[i] = new Dog( Integer.toString(i) );

// Walk dogs.
while( !dogsFinished )
{
   for( Dog d : pack )
   {
      d.walk();

      if( d.isThirsty )

         d.drink();

   }

   // See if all the dogs are finished with their walks.
   // Walk until all dogs are beyond the max distance.
   count = 0;
  for( Dog d : pack )
   {
      if( d.distanceWalked > MAX_WALK_DISTANCE )
         count++;
   }

  if( count == pack.length )
     dogsFinished = true;
}

// Display results.
for( Dog d : pack )
   System.out.println( d );
```

toString() Method

1. Notice above, that we are printing an object, *d*. What does this mean, to print an object? We can define what it means by providing a *toString()* method that returns whatever we want. This method is called on an object automatically when you ask to print it (There is actually a lot behind the scenes here including inheritance, overriding methods, and the *String.valueOf()* method). A *toString()* method is a very common way to display the data in an object, *i.e.* to display the state of the object.

```
System.out.println( d );
```

Thus, we add the *toString()* method to the Dog class.

```
public String toString()
{
     double numMiles = this.distanceWalked / 1260.0;

     String msg = String.format( "Dog: '%s' walked: %.3f miles" +
                              " and stopped to drink water %d times.",
                              this.name, numMiles, this.numDrinks );

     return msg;
}
```

If you do not provide a *toString()* method, the print statement above will simply print the memory address of the object. We can call the *toString()* method directly if we want to:

```
System.out.println( d.toString() );
```

Overloaded Methods

1. We can *overload* methods just as we did in the last chapter:

```
void bark()
{
   System.out.println( "bark" );
}

void bark( int n )
{
   for( int i=0; i<n; i++ )
      System.out.println( "BARK" );
}
```

Passing Objects to Methods

1. Let's consider a situation where we have dog and cat objects. And, when a dog encounters a cat, it growls at it. Thus, we may write a *Cat* class like this:

```
public class Cat
{
      public String name;

      public Cat( String catName )
      {
         name = catName;
      }
}
```

And we can add a method to the *Dog* class:

```
public void encounterCat( Cat cat )
{
   System.out.println( name + ", growls at " + cat.name );
}
```

Finally, we can test the new methods with the code shown below in *main()*. Note that this *main()* could be in either class, *Cat* or *Dog*, or both. Regardless, we compile each file and then run one with a *main*.

```
Dog d = new Dog( "Spot" );
Cat c = new Cat( "Felix" );

d.bark();
d.bark(3);
d.encounterCat( c );
```

Which produces the output:

```
bark
BARK
BARK
BARK
Spot, growls at Felix
```

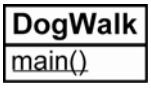
Creating a *Driver* Class

1. Now let's consider another way to run a program. So far, we have always put our *main()* in the class we are writing. In the previous example, we had two classes, *Cat* and *Dog* each saved in separate files. I put the main in the Dog class, but the exact same code can also be in the Cat class. As we start to work with more classes (CS 1302), it is frequently useful to have a separate class to *start things up*. Sometimes we call this a *driver* class. For instance, consider the original problem from this section, walking a dog. We could make a driver class, say *DogWalk* that looks like this:

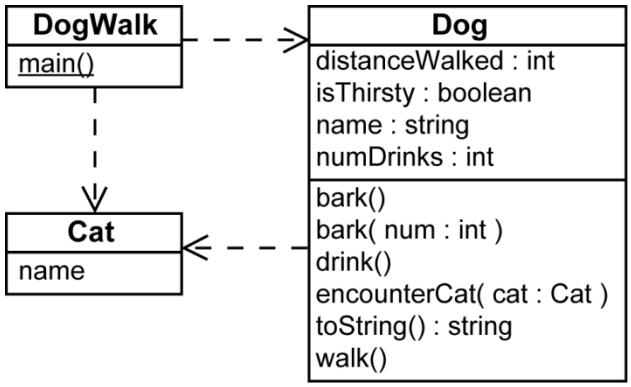
```
public class DogWalk
{
      public static void main(String[] args)
      {
         final int MAX_WALK_DISTANCE = 2*1260;

         Dog d = new Dog("Max");
         Cat c = new Cat( "Felix" );

         d.bark();
         d.bark(3);
         d.encounterCat( c );

         while( d.distanceWalked < MAX_WALK_DISTANCE )
         {
            d.walk();

            if( d.isThirsty )

               d.drink();
         }

         System.out.println( d );
      }
}
```

   Finally, to run the program, we just run the DogWalk class.

2. A class diagram for the *DogWalk* class is shown below. Notice that in UML, static methods are underlined. Notice also that a class diagram does not show variables that are declared *inside* a method.



A class diagram for the whole program is shown below. The dashed lines indicate *dependency*. In other words, the DogWalk class *depends on* (or *uses*) the Dog class and the Cat class, while the Dog class depends on the Cat class. As systems grow, there are many of these dependencies. Usually, we only show important dependencies on a class diagram (for now, all of them) because we have other things to put in a class diagram that we will learn later, and next semester.



Associations – *has a* relationships

In object oriented programming, many times, one object *has* another object. For instance a *person has a dog*. This is referred to as a "*has a*" relationship and means that the *Person class has a property which is a Dog object*:

```
public class Person
{
	public String name;
	public Dog dog;

	public Person( String personName )
	{
	   name = personName;
	}
}
```

So that we can write code like this in a main():

```
Person p = new Person( "Anne" );

Dog d = new Dog("Max");

p.dog = d;

p.dog.bark(5);
```

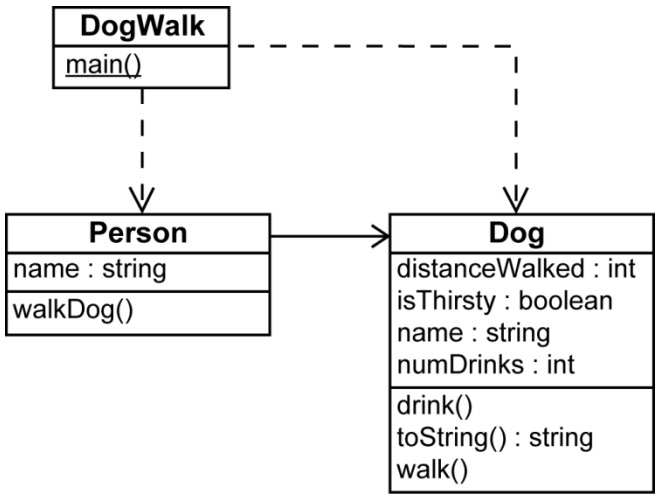
Now, returning to the original problem (the simulation), we may want the person to walk the dog, so we will add a *walkDog* method to the Person class.

```
public class Person
{
      public String name;
      public Dog dog;

      public Person( String personName )
      {
         name = personName;
      }

      public void walkDog()
      {
         final int MAX_WALK_DISTANCE = 2*1260;

         while( dog.distanceWalked < MAX_WALK_DISTANCE )
         {
            dog.walk();

            if( dog.isThirsty )

               dog.drink();
         }
      }
}
```

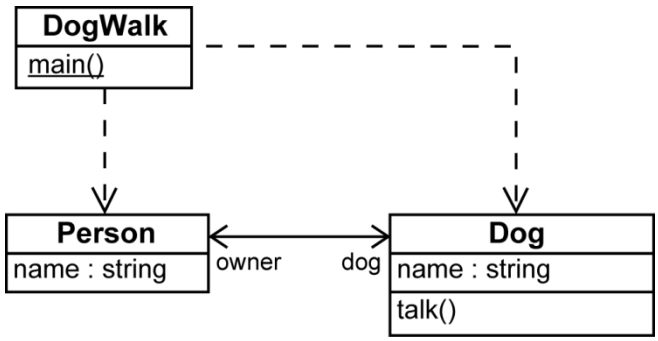
Now, to do the simulation:

```
public static void main(String[] args)
{
   Person p = new Person( "Anne" );

   Dog d = new Dog("Max");

   p.dog = d;

   p.walkDog();

   System.out.println( d );
}
```

A class diagram for this system is shown below. Notice the solid-line arrow between Person and Dog. In UML, this is called an *association*. It means that the Person class has an instance variable of type Dog. Notice that by convention we do not show this instance variable (even when it is public) in the Person class (*e.g.* the solid-line takes care of that for us). The direction of the arrow indicates *navigability*. Since the arrow points from a Person to a Dog, given a Person object, we can always find the associated Dog. However, given a Dog, we cannot find out it's owner (*e.g.* there is no arrow from Dog to person).



In other situations, it may be important for a Dog to know who his owner is:



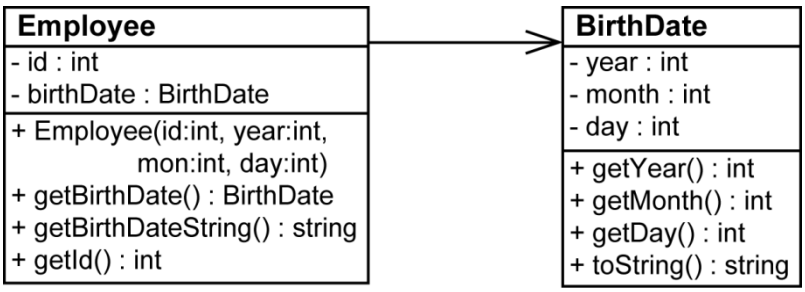

```
public class Person
{
      public String name;
      public Dog dog;

      public Person( String personName )
      {
         name = personName;
      }
}
```

```
public class Dog
{
      public String name;
      public Person owner;

      public Dog( String dogName )
      {
         name = dogName;
      }

      void talk()
      {
          String msg = String.format( "My name is '%s' and " +
                                      "my owner's name is '%s'.",
                                      this.name, owner.name );

         System.out.println( msg );
     }
}
```

```
public static void main(String[] args)
{
   Person p = new Person( "Anne" );

   Dog d = new Dog("Max");
   d.owner = p;
   p.dog = d;

   p.dog.talk();

}
```

Association – The "has a" Relationship

As we have seen, a class can have a data field that references an object from another (or the same) class. For instance, an Employee class may have a data field that references a *BirthDate* object. Consider the two classes:




```
class Employee
{
     private int id;
     private BirthDate birthDate;        // Encapsulated

     Employee( int idNum, int year, int month, int day )
     {
        id = idNum;
        birthDate = new BirthDate( year, month, day );
     }

     public BirthDate getBirthDate()     // Getter
     {
        return birthDate;
     }

    public String getBirthDateString()
     {
        return birthDate.toString();     // Delegation
     }

     public int getId() { return id; }   // Getter
}
```

```
class BirthDate                              // Immutable class
{
     private int year;
     private int month;
     private int day;

     BirthDate( int y, int m, int d )
     {
        this.year = y; this.month = m;   this.day = d;
     }
     public int getYear() { return year; }

     public int getMonth() { return month; }

     public int getDay() { return day; }

     public String toString()
     {
        return month + "/" + day + "/" + year;
     }
}
```

Notice in this example that we *create* the association in the constructor and that this association can never be changed (there is a *getter* but no *setter*). Further, the *BirthDate* object itself, although it can be accessed, can itself never be changed; it is immutable.

So, we can write *client* code like this:

```
Employee e = new Employee( 39873, 88, 4, 7 );

BirthDate bd = e.getBirthDate();

System.out.println( bd.getYear() );
```

Notice that the *Employee* class has a public method, *getBirthDateString()* which returns the birth date as a string. This is a *convenience method*. It is provided so that the programmer can get the birth date *simply.* The way it accomplishes this is called *delegation*; it *delegates* to the *BirthDate* object. Thus, we can write code like this:

```
System.out.println( e.getBirthDateString() );
```

So, the class provides a quick and easy way to get the birth date, *getBirthDateString* and it also provides access to the entire *BirthDate* object if the client needs more control over formatting:

```
System.out.println( bd.getMonth() + "-" +
                    bd.getDay() + "-" +
                     bd.getYear() );
```

In the constructor for the *BirthDate* class, we the java reference *this*, *this.year = y*, etc. This will be explained later. In this example, it is not needed (could be completely removed), but is included to get us used to seeing it (because we will). To be explained!