Introduction to Classes

In this lesson, an explanation of the basics about classes in Java- what are they and how to define them is provided.

We'll cover the following

Definition

`Classes` are the building blocks of programs built using the object-oriented methodology. Objects have certain similar traits - state and behavior. This commonality is provided in a blueprint or template for the instantiation of all similar objects. This blueprint is known as a *class*.

Body of class

```
class className {

  int integer_one;
  String string_one;

}
```

A *Keyword* `class` is used with every *declaration* of class followed by the name of the class. You can use any *className* as you want.

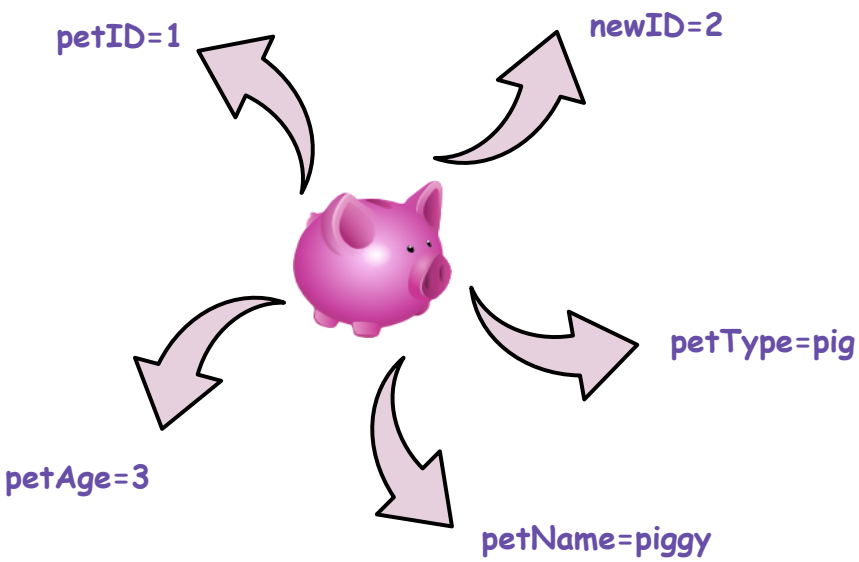
Fundamentals

Every class in Java can be composed of the following elements:

- `fields, member variables or instance variables` - These are variables that hold data specific to a particular object. For each object of a class, there is one field.
- `member methods or instance methods` - These perform operations on objects and are defined within the class.
- `static or class fields` - These fields are common to any object within the same class. They can only exist once in the class irrespective of the total number of objects in the class.
- `static or class methods` - These methods do not affect a specific object in the class.
- `inner classes` - At times a class will be defined within a class if it is a subset of another class. The class contained within another is the **inner** class.

Now that we understand these basic concepts within classes, let's look at an example of a class and see if we can understand them better.

Example of a pet class



A visual example of the Pet class

```
class Pet {

    private static int newID;
    private int petID;
    private String petType;
    private String petName;
    private int petAge;

    public Pet(String type, String name, int age) {
        petType = type;
        petID = newID;
        petName = name;
        petAge = age;

        newID = newID + 1;
    }

    public String getName() {
        return petType;
    }

    public static int getNewID() {
        return newID;
    }

    public void printPetDetails() {
        System.out.println("Pet ID: " + petID);
        System.out.println("Pet Type: " + petType);
        System.out.println("Pet Name: " + petName);
        System.out.println("Pet Age: " + petAge);
    }

}

class PetList {
    public static void main(String[] args) {

        Pet myDog = new Pet("dog", "Ruffy", 3);
        myDog.printPetDetails();

        Pet newcat = new Pet("cat", "Princess", 2);
        newcat.printPetDetails();
    }
}
```

Explanation

For this example we have taken a **Pet class**. In this class we have five variables:

- `petType`: This determines the type of animal that the particular pet is
- `petID`: This is the identification number given to an individual pet

- `petName`: This is the name that the pet will be assigned

- `petAge`: This is the integer value of the age of the pet
- `newID`: This is a **static variable**, hence only one copy of this field will exist, no matter how many Pets are created.

An instance of a *pet*, say, a **dog** named **Ruffy**, would be an **object**. So would a **cat** named **Princess**. Hence, you can have *multiple* instances of a *class*, just like you can have *multiple* pets.

Properties, that is the variables defined in the class, are like **"inner variables"** of each *object* made of type `Pet`.

To **declare** objects of a class we should follow the format on Lines 38 and 41.

We used the **dot** operator to access members of a class *object*. See how in the above example, the methods are called in Lines 39 and 41.

Private members

As you can see above, we have used the word `private` before *declaring* the class variables. The *private members* can only be referenced within the definitions of member methods. If a program tried to access `private` *variables* directly it will get a **compiler error**.

> **Note:** Private members can be *variables* or *methods*.

Try running the code below. It will give an error when you try to compile it, as in the code, private members of the class are being accessed directly.

```
class Pet {
    private String petName;
    private String petType;

    public Pet() {};
}

class PetList {

    public static void main(String[] args) {
        Pet myDog = new Pet();
        System.out.println(myDog.petType);
    }
}
```

The code within the same class can access private members. See the code given below!

```
class Pet {
    private String petName;
    private String petType;

    public Pet() {};

    public static void main(String[] args) {
        Pet myDog = new Pet();
        System.out.println(myDog.petType);
    }
}
```

A method can refer to a private variable of the class using the same syntax as if it were a local variable defined in the method.

> **Quick Question:** Why we need to keep the members priavte?

Hide Hint

The Object Oriented paradigm takes inspiration from the fact that the state of a real world object is changed by sending messages to it, instead of directly fiddling with an object's state. Hence the motivation to keep instance variables private.

Public members

The keyword `public` identifies members of a class that can be accessed from outside of the class. Look at the **methods** in the `Pet Class` example. See how these are being called in the `main()` method without any compilation errors.

Now that we have understood classes and the basic types of elements they can

Now that we have understood, classes and the basic types of elements they can have, the next lesson will delve deeper into class member methods.