Computer Programming 2

Module 5: Classes and Objects

Name (LN,FN,MN): _____ Program/Yr/Block: _____

I. Introduction

This module will cover fundamental concepts of object-oriented programming, classes and objects. One major difference between objects and class is in the way attributes and methods are treated in objects and classes. A class is a definition about objects; the attributes and methods in a class are thus declarations that do not contain values. However, objects are created instances of a class. Each has its own attributes and methods. The values of the set of attributes describe the state of the objects.

Please feel free to use other references and resources which you think will greatly help you in your programming journey. Enjoy coding!

II. Learning Objectives

After completing this module, you should be able design and write programs that use the following concepts:

1. Classes
2. Objects
3. Access Modifiers
4. Constructors
5. Methods and method overloading

III. Topics and Key Concepts

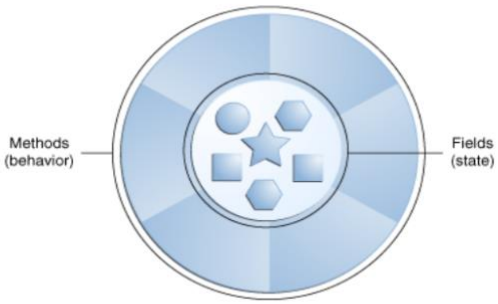
A. What is an Object?

Objects are key to understanding object-oriented technology. Look around right now and you'll find many examples of real-world objects: your dog, your desk, your television set, your bicycle.

Real-world objects share two characteristics: They all have state and behavior. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.

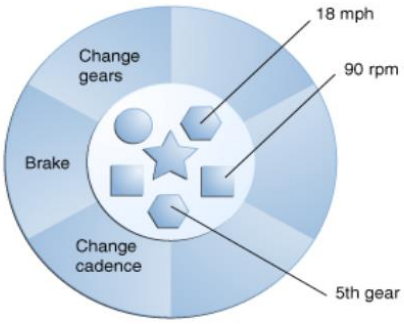
Software objects are conceptually similar to real-world objects: they too consist of state and related behavior. An object stores its state in **fields**

(variables in some programming languages) and exposes its behavior through **methods** (functions in some programming languages). Methods operate on an object's internal state and serve as the primary mechanism for object-to-object communication. Hiding internal state and requiring all interaction to be performed through an object's methods is known as data encapsulation — a fundamental principle of object-oriented programming.



A software object.

Consider a bicycle, for example:



A bicycle modeled as a software object.

By attributing state (current speed, current pedal cadence, and current gear) and providing methods for changing that state, the object remains in control of how the outside world is allowed to use it. For example, if the bicycle only has 6 gears, a method to change gears could reject any value that is less than 1 or greater than 6.

Bundling code into individual software objects provides a number of benefits, including:

Modularity: The source code for an object can be written and maintained independently of the source code for other objects. Once created, an object can be easily passed around inside the system.

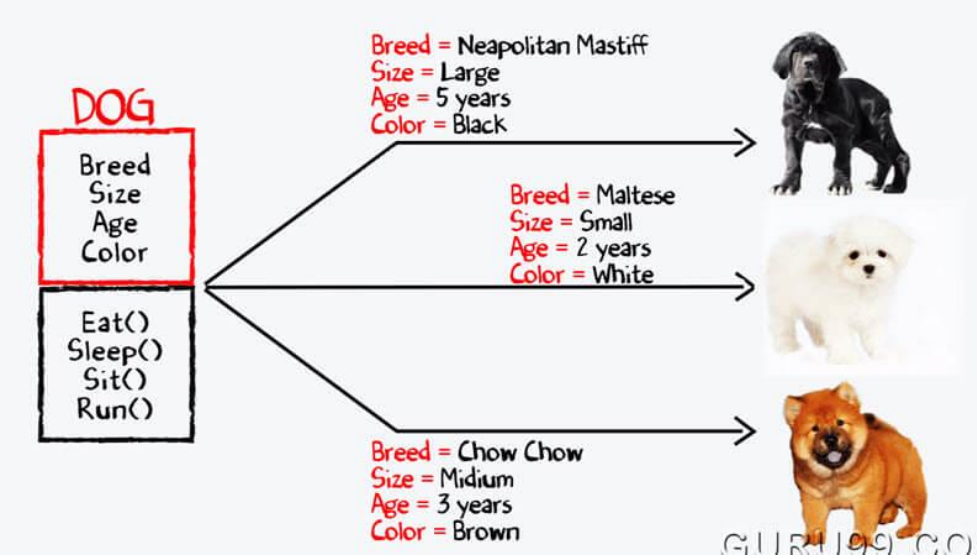
Information-hiding: By interacting only with an object's methods, the details of its internal implementation remain hidden from the outside world.

Code re-use: If an object already exists (perhaps written by another software developer), you can use that object in your program. This allows specialists to implement/test/debug complex, task-specific objects, which you can then trust to run in your own code.

Pluggability and debugging ease: If a particular object turns out to be problematic, you can simply remove it from your application and plug in a different object as its replacement. This is analogous to fixing mechanical problems in the real world. If a bolt breaks, you replace it, not the entire machine.

B. What is a Class?

In the real world, you'll often find many individual objects all of the same kind. There may be thousands of other bicycles in existence, all of the same make and model. Each bicycle was built from the same set of blueprints and therefore contains the same components. In object-oriented terms, we say that your bicycle is an instance of the class of objects known as bicycles. A class is the blueprint from which individual objects are created.



Difference between a class and its instances (objects)

The following Bicycle class is one possible implementation of a bicycle:

```
class Bicycle {

    int cadence = 0;
    int speed = 0;
    int gear = 1;

    void changeCadence(int newValue) {
         cadence = newValue;
    }

    void changeGear(int newValue) {
         gear = newValue;
    }

    void speedUp(int increment) {
         speed = speed + increment;
    }

    void applyBrakes(int decrement) {
         speed = speed - decrement;
    }

    void printStates() {
         System.out.println("cadence:" +
             cadence + " speed:" +
             speed + " gear:" + gear);
    }
}
```

The syntax of the Java programming language will look new to you, but the design of this class is based on the previous discussion of bicycle objects. The fields cadence, speed, and gear represent the object's state, and the methods (changeCadence, changeGear, speedUp etc.) define its interaction with the outside world.

You may have noticed that the Bicycle class does not contain a main method. That's because it's not a complete application; it's just the blueprint for bicycles that might be used in an application. The responsibility of creating and using new Bicycle objects belongs to some other class in your application.

Here's a BicycleDemo class that creates two separate Bicycle objects and invokes their methods:

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

        // Create two different
        // Bicycle objects
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle();

        // Invoke methods on
        // those objects
        bike1.changeCadence(50);
        bike1.speedUp(10);
        bike1.changeGear(2);
        bike1.printStates();

        bike2.changeCadence(50);
        bike2.speedUp(10);
        bike2.changeGear(2);
        bike2.changeCadence(40);
        bike2.speedUp(10);
        bike2.changeGear(3);
        bike2.printStates();
    }
}
```

The output of this test prints the ending pedal cadence, speed, and gear for the two bicycles:

```
cadence:50 speed:10 gear:2
cadence:40 speed:20 gear:3
```

Declaring Classes

You've seen classes defined in the following way:

```
class MyClass {
    // field, constructor, and
    // method declarations
}
```

This is a class declaration. The class body (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class: constructors for initializing new objects, declarations for the fields that provide the state of the class and its objects, and methods to implement the behavior of the class and its objects.

The preceding class declaration is a minimal one. It contains only those components of a class declaration that are required. You can provide more information about the class, such as the name of its superclass, whether it

implements any interfaces, and so on, at the start of the class declaration. For example,

```
class MyClass extends MySuperClass implements YourInterface {
    // field, constructor, and
    // method declarations
}
```

means that MyClass is a subclass of MySuperClass and that it implements the YourInterface interface.

You can also add modifiers like public or private at the very beginning—so you can see that the opening line of a class declaration can become quite complicated. The modifiers public and private, which determine what other classes can access MyClass, are discussed later in this module. The lesson on interfaces and inheritance will explain how and why you would use the extends and implements keywords in a class declaration. For the moment you do not need to worry about these extra complications.

In general, class declarations can include these components, in order:

1. Modifiers such as public, private, and a number of others that you will encounter later.
2. The class name, with the initial letter capitalized by convention.
3. The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
4. A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
5. The class body, surrounded by braces, {}.

C. Member Variables and Access Modifiers

There are several kinds of variables:

- Member variables in a class—these are called **fields.**
- Variables in a method or block of code—these are called **local variables**.
- Variables in method declarations—these are called **parameters.**

The Bicycle class uses the following lines of code to define its fields:

```
public int cadence;
public int gear;
public int speed;
```

Field declarations are composed of three components, in order:

1. Zero or more modifiers, such as public or private.
2. The field's type.
3. The field's name.

The fields of Bicycle are named cadence, gear, and speed and are all of data type integer (int). The public keyword identifies these fields as public members, accessible by any object that can access the class.

Access Modifiers

The first (left-most) modifier used lets you control what other classes have access to a member field. For the moment, consider only public and private. Other access modifiers will be discussed later.

- **public** modifier—the field is accessible from all classes.
- **private** modifier—the field is accessible only within its own class.

In the spirit of encapsulation, it is common to make fields private. This means that they can only be directly accessed from the Bicycle class. We still need access to these values, however. This can be done indirectly by adding public methods that obtain the field values for us:

```
public class Bicycle {

    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public int getCadence() {
        return cadence;
    }

    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public int getGear() {
        return gear;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public int getSpeed() {
        return speed;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

    public void speedUp(int increment) {
        speed += increment;
    }
}
```

All variables must have a type. You can use primitive types such as int, float, boolean, etc. Or you can use reference types, such as strings, arrays, or objects.

ACCESS SPECIFIER	ACCESSIBLE BY CLASS DEFINITION	ACCESSIBLE BY SUBCLASS DEFINITION	ACCESSIBLE BY REST OF PACKAGE	ACCESSIBLE BY REST OF WORLD
private	yes	no	no	no
protected	yes	yes	yes	no
public	yes	yes	yes	yes
none i.e. package	yes	no	yes	no

Java Access Modifiers

All variables, whether they are fields, local variables, or parameters, follow the same naming rules and conventions.

In this module, be aware that the same naming rules and conventions are used for method and class names, except that

- the first letter of a class name should be capitalized, and
- the first (or only) word in a method name should be a verb.

D. Methods

Here is an example of a typical method declaration:

```
public double calculateAnswer(double wingSpan, int numberOfEngines,
                              double length, double grossTons) {
    //do the calculation here
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, (), and a body between braces, {}.

More generally, method declarations have six components, in order:

1. Modifiers—such as public, private, and others you will learn about later.
2. The return type—the data type of the value returned by the method, or void if the method does not return a value.
3. The method name—the rules for field names apply to method names as well, but the convention is a little different.
4. The parameter list in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, (). If there are no parameters, you must use empty parentheses.

5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

- Every class can have a set of methods associated with it which allow functionality for the class.
  - **Accessor** method
    - –Often called "getter" method.
    - –Returns the value of a specific instance variable.
  - **Mutator** method
    - –Often called "setter" method.
    - –Changes or sets the value of a specific instance variable.
  - **Functional** method
    - –Returns or performs some sort of functionality for the class.

Two of the components of a method declaration comprise the **method signature**—the method's name and the parameter types.

The signature of the method declared above is:

```
calculateAnswer(double, int, double, double)
```

Although a method name can be any legal identifier, code conventions restrict method names. By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc. In multi-word names, the first letter of each of the second and following words should be capitalized. Here are some examples:

```
run
runFast
getBackground
getFinalData
compareTo
setX
isEmpty
```

Typically, a method has a unique name within its class. However, a method might have the same name as other methods due to method overloading.

E. Method Overloading

The Java programming language supports overloading methods, and Java can distinguish between methods with different method signatures. This means that methods within a class can have the same name if they have different parameter lists.

Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type. It is cumbersome to use a new name for each method—for

example, drawString, drawInteger, drawFloat, and so on. In the Java programming language, you can use the same name for all the drawing methods but pass a different argument list to each method. Thus, the data drawing class might declare four methods named draw, each of which has a different parameter list.

```
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

Overloaded methods are differentiated by the number and the type of the arguments passed into the method. In the code sample, draw(String s) and draw(int i) are distinct and unique methods because they require different argument types.

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.

Overloaded methods should be used sparingly, as they can make code much less readable.

F. Constructors

A class contains constructors that are invoked to create objects from the class blueprint. Constructor declarations look like method declarations—except that they use the name of the class and have no return type. For example, Bicycle has one constructor:

```
public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}
```

To create a new Bicycle object called myBike, a constructor is called by the new operator:

```
Bicycle myBike = new Bicycle(30, 0, 8);
```

`new Bicycle(30, 0, 8)` creates space in memory for the object and initializes its fields.

Although Bicycle only has one constructor, it could have others, including a no-argument constructor:

```
public Bicycle() {
    gear = 1;
    cadence = 10;
    speed = 0;
}
```

`Bicycle yourBike = new Bicycle();` invokes the no-argument constructor to create a new `Bicycle` object called `yourBike`.

Both constructors could have been declared in Bicycle because they have different argument lists. As with methods, the Java platform differentiates constructors on the basis of the number of arguments in the list and their types. You cannot write two constructors that have the same number and type of arguments for the same class, because the platform would not be able to tell them apart. Doing so causes a compile-time error.

You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of Object, which does have a no-argument constructor.

IV. Teaching and Learning Materials Resources

- PC Computer | Laptop | Android Phone
- Gordon College LAMP
- Google Meet
- Facebook Messenger

V. Learning Tasks

A. Explore (50 max points)

Use your preferred IDE or editor and follow the steps below:

1. Create a class called Dog that does not have a main method.
2. Include the following instance fields in the dog class:
   - String name
   - String breed
   - String barkNoise – "Woof"
   - Double weight
3. Create a constructor that creates a dog object using parameters for the name, breed and weight fields. Assign the instance fields to these values.
4. Create a second constructor that accepts values for all instance fields as parameters and assigns the corresponding fields to those values.
5. Create Accessor and Mutator (getter and setter) methods for the following fields:
   - name
   - breed
   - Weight

   Do not create one for the barkNoise field!
6. Add a functional method called bark that will display the value of the barkNoise field to the console.
7. Add an overloaded functional method called bark that will accept a String parameter for the noise the dog will make when it barks.
8. Create a class called AnimalTester that contains a main method.
9. Create a dog1 object using the 3 parameter constructor.

Name	Ace
Breed	Beagle
weight	45.6

Name	Bailey
Breed	Boerboel
Bark noise	arf-arf
weight	80.2

10. Create a dog2 object using the 4 parameter constructor.
11. Use the instance field getter methods to produce the following output:

```
Dog name  : Bailey
Dog breed : Boerboel
Bark noise: arf-arf
Dog weight: 80.2
```

(Additional Points: Inheritance)

12. Create an Animal class (superclass) and a Fish class (subclass, same as Dog).
13. They would like to store the colour of all of their animals. All animals must have their colour stored.

14. You cannot store any fish objects in your system unless all of the required values have been provided. Create a fish constructor that reflects this.
15. Identify the common fields/methods and place them in the superclass. The fields/methods that are specific to each animal type (dog, fish) should be placed in the appropriate class.
16. Display both the fish and dog values to the console.
17. A fish can be either a cold or heated water variety.
18. Update your driver class (AnimalTester) to create these objects:

Breed	Goldfish
Water type	cold
colour	red

Name	Bailey
Breed	Boerboel
Bark noise	arf-arf
weight	80.2
colour	brown

Your code: Animal.java (provide screenshot)

Your code: Dog.java (provide screenshot)

Your code: Fish.java (provide screenshot)

Your code: AnimalTester.java (provide screenshot)

Sample Run: java AnimalTester (provide screenshot)

B. Explain (50 max points)

1. Differentiate a class from an object

Class	Object

2. What is encapsulation and how do you implement it in your code?

3. Using your own words, define the following terms

State	
Behavior	
Instance	
Instance Fields	
Method	
Constructor	
Method overloading	

4. Based on your observation, list down at least five (5) states and five (5) behaviors of the following objects:

A. Bank Account

State	Behavior

B. Library

State	Behavior

C. Engage (50 max points)

1. Consider the following class definition and then answer the question below:

```
public class Account {
   int balance;

   public Account(int openingBalance) {
      balance = openingBalance;
   }

   public void deposit(int pesos) {
      balance = balance + pesos;
   }

   public boolean withdraw(int pesos) {
      if (balance < pesos)
         return false;
      else {
         balance = balance - pesos;
         return true;
      }
   }

   public boolean transfer (int pesos, Account destination) {
      if (withdraw(pesos)) {
         destination.deposit(pesos);
         return true;
      }else
         return false;
   }
```

```
    public String toString() {
       return ("Php " + balance + ".00");
    }

   public static void main(String args[]){
        Account alice = new Account(100);
        Account bob = new Account(100);
        alice.deposit(25);
        System.out.println("Alice has " + alice); // Line 1 toString() method is invoked
        System.out.println("Bob can withdraw  Php 125.00? " + bob.withdraw(125)); // Line 2
        System.out.println("Bob can withdraw  Php 25.00? " + bob.withdraw(25)); // Line 3
        System.out.println("Bob has " + bob); // Line 4 toString() method is invoked
        Account charlie = alice;
        System.out.println("Charlie can withdraw Php 25.00? " + charlie.withdraw(25)); // Line 5
        System.out.println("Charlie has " + charlie); // Line 6 toString() method is invoked
        System.out.println("Alice has " + alice); // Line 7 toString() method is invoked
        alice.transfer(50, charlie);
        System.out.println("After transfer..."); // Line 8
        System.out.println("Charlie has " + charlie); //  Line 9 toString() method is invoked
        System.out.println("Alice has " + alice); // Line 10 toString() method is invoked
   }
}
```

What output would be printed as the result of executing the following code?

Line 1 _____
Line 2 _____
Line 3 _____
Line 4 _____
Line 5 _____
Line 6 _____
Line 7 _____
Line 8 _____
Line 9 _____
Line 10 _____

2. The Library Project
   The libraries of SmallTownX need a new electronic rental system, and it is up to you to build it. SmallTownX has two libraries. Each library offers many books to rent. Customers can print the list of available books, borrow, and return books.

   We provide two classes, Book and `Library`, that provide the functionality for the book database. You must implement the missing methods to make these classes work.

Step 1: Implement Book
First, we need a class to model books. Start by creating a class called Book. Copy and paste the skeleton code given. This class defines methods to get the title of a book, find out if it is available, borrow the book, and return the book. However, the skeleton that we provide is missing the implementations of the methods. Fill in the body of the methods with the appropriate code. The main method tests the methods. When you run the program, the output should be:

```
Title (should be The Da Vinci Code): The Da Vinci Code
Rented? (should be false): false
Rented? (should be true): true
Rented? (should be false): false
```

Hint: Look at the main method to see how the methods are used, then fill in the code for each method.

Step 2: Implement Library
Next, we need to build the class that will represent each library, and manage a collection of books. All libraries have the same hours: 9 AM to 5 PM daily. However, they have different addresses and book collections (i.e., arrays of Book objects).

Create a class called Library. Copy and paste the skeleton code given. We provide a main method that creates two libraries, then performs some operations on the books. However, all the methods and member variables are missing. You will need to define and implement the missing methods. Read the main method and look at the compile errors to figure out what methods are missing.

Notes

- Some methods will need to be static methods, and some need to be instance methods.
- Be careful when comparing Strings objects. Use string1.equals(string2) for comparing the contents of string1 and string2.
- You should get a small part working at a time. Start by commenting the entire main method, then uncomment it line by line. Run the program, get the first lines working, then uncomment the next line, get that working, etc.
- You must not modify the main method.

The output when you run this program should be similar to the following:

```
Library hours:
Libraries are open daily from 9am to 5pm.

Library addresses:
10 Main St.
228 Liberty St.

Borrowing The Lord of the Rings:
You successfully borrowed The Lord of the Rings
Sorry, this book is already borrowed.
Sorry, this book is not in our catalog.

Books available in the first library:
The Da Vinci Code
Le Petit Prince
A Tale of Two Cities

Books available in the second library:
No book in catalog

Returning The Lord of the Rings:
You successfully returned The Lord of the Rings

Books available in the first library:
The Da Vinci Code
Le Petit Prince
A Tale of Two Cities
The Lord of the Rings
```

VI. References

- Oracle. nd. “Oracle Java Documentation”. https://docs.oracle.com/javase/tutorial/
- Oracle Academy. 2020. “Java Programming Instructor Resources”. https://academy.oracle.com
- Sedgewick Robert, Princeton University, Wayne, Kevin. 2017. “Introduction to Programming in Java: An Interdisciplinary Approach, 2nd Edition”. https://introcs.cs.princeton.edu/java/home/