303.5.2
Object-Oriented Programming
Java - Encapsulation Inheritance

Lesson 1

Encapsulation

One of the pillars of Object-Oriented Programming

Learning Objectives:

By the end of lesson, learners will be able to:

- Define the Encapsulation concept.
- Describe the bundling of data and methods used to operate the data into one unit.
- Utilize getters, setters, and keywords in Java.
- Describe constructors, constructor overloading, and access modifiers.
- Demonstrate the use of contractors, contractor overloading, and access modifiers in in Java programming.

Outline

- Topic 1: Introduction to Encapsulation
- Topic 1a: Getter and Setter Methods
- Topic 1b: The “this” Keyword in Java
- Hands-On Lab

Topic 1: Introduction to Encapsulation

Data Hiding:

In Encapsulation, the class variables will be hidden from other classes, and can be accessed only through the methods of their current class. Therefore, encapsulation is also referred to as *data hiding*.

- Encapsulation is a process that binds together the data and the functions that manipulate the data, keeping both safe from outside interference and misuse.
- In order to achieve Encapsulation in Java, follow certain steps as proposed below:
  - **Rule of Thumb:** Class variables in a class must be either **private** or **protected**, unless there is a good reason for them not to be.
  - Methods can be created, which return or change the current value of a variable, **but the variable itself should not be accessible outside the class.** Usually, we refer to these as **"getter"** and **"setter" methods.**

Topic 1a: Getter and Setter Methods

- Setter and Getter methods are **public**, hence visible to other classes.
- For each Class variable (private), a getter method (getters) returns its value, while a setter method (setters) sets or updates its value. Given this, getter and setter methods are also known as *accessors* and *mutators*, respectively.
- By convention, getters start with the word **"*get*"** and setters with the word **"*set*,"** followed by a variable name. In both cases, the first letter of the variable's name is **capitalized**:
  - `getXxx()` and `setXxx()`, where `Xxx` is the name of the variable.

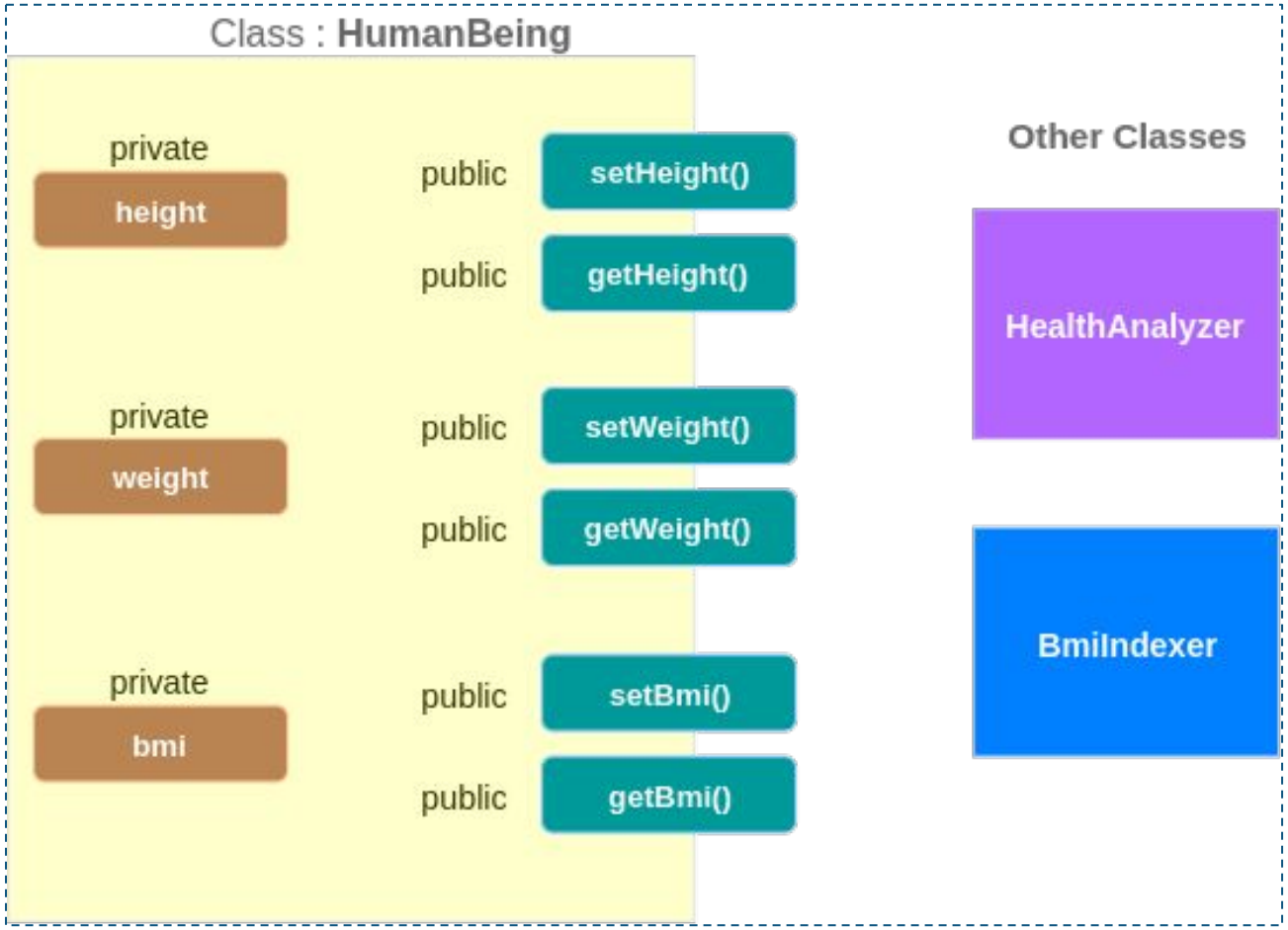
Please view our **Wiki** documentation for more information about **Getter and Setter Methods**.

Example:

```
public class Vehicle {
  private String color;
  // Getter
  public String getColor() {
    return color;
  }
  // Setter
  public void setColor(String c) {
    this.color = c;
  }
}
```

Encapsulation Illustration

The below diagram provides an illustration about **Encapsulation** in Java.



- Class Variables (height, weight, and bmi) in the example are declared **private;** hence, they are not visible to other classes.
- For each variable, there is a **setter** method and **getter** method, which sets a value to the variable and gets the value of the variable respectively. For example, for variable height, the setter method is **setHeight().**, For variable height, the getter method is **getHeight()**.
- Setter and Getter methods are public, and visible to other classes.

Topic 1b: The "**this**" Keyword in Java

- You can use the keyword "this" to refer to ***this*** instance inside a class definition.
- One of the main uses of the "this" keyword is to resolve ambiguity.

Example

```
public class Circle {
   double radius;
            // Class variable called "radius"
   public Circle(double radius) { // Constructor and method's parameter also called "radius"

      this.radius = radius;
         // "radius = radius" does not make sense!
         // "this.radius" refers to this instance's member variable
         // "radius" resolved to the method's parameter.
   }
   ...
}
```

Please view our **Wiki** documentation for more explanation and details about **this keyword**.

Encapsulation Example


```
public class Person{
        private String name;
        private int age;
        private String company;
        private String job;
        private String hobby;
        public Person()
        {
        }
    public Person(String name,  int age, String company,
String job, String hobby) {
        this.name = name;
        this.age = age;
        this.company = company;
        this.job = job;
        this.hobby = hobby;
    }
    public String getName() {
        return name;    }
    public void setName(String name) {
        this.name = name; }
    public int getAge() {
        return age;    }
    public void setAge(int age) {
        this.age = age;    }
    public String getCompany() {
        return company;
    }
```

```
public void setCompany(String company) {
        this.company = company;
    }
    public String getJob() {
        return job;    }

    public void setJob(String job) {
        this.job = job;    }

    public String getHobby() {
        return hobby;    }

    public void setHobby(String hobby) {
        this.hobby = hobby;    }

    public void talkaboutYourself () {
        System.out.println("Hi, I'm" + this.name + ".");
        System.out.println("I'm " + this.age + " years old. ");
        System.out.println("I work at " + this.company + "as a "
+ job + ",");
        System.out.println("When I get some free time, I like to
" + this.hobby + ".");
    }
}
```

Encapsulation Example (continued)

Create a new class named **Mytest. This** class will be our ***main*** class or ***entry point*** of the project. Write the code below:

```
public class Mytest {
   public static void main(String[] args) {
        Person mike =new Person ();
        mike.setName(" Mike") ;
        mike.setAge(22);
        mike.setCompany("Per Scholas") ;
        mike.setJob("Technical Instructor");
        mike.setHobby ("learn about Financial Engineering") ;
        mike.talkaboutYourself();
   }
}
```

Instantiating the Class

Output

Hi, I'm Mike.
I'm 22 years old.
I work at Per Scholas as a Technical Instructor.
When I get some free time, I like to learn about Financial Engineering.

Hands-On Lab

Complete **LAB - 303.5.1 Encapsulation in Java**. You can find this Lab on Canvas under the **Guided Lab section.**

If you have any technical questions while performing the lab activity, ask your instructors for assistance.

Practical Assignment: Encapsulation

From the following table, create a class named ***Student*** *and declare* ***Class variables*** that can hold the data for a single student:

Class Variables	Student ID	First Name	Last Name	Student Email	Student Phone
Data	0001	Michael	Gabriel	mgabriel@perscholas.org	123-456-7890
Data	0002	Bairon	Vasquez	bvasquez@perscholas.org	123-456-7891

Note: Make sure that you use **Encapsulation**, and include a **Constructor**. You can pass data using the setter() method or constructor.

Summary

In this section, we discussed Encapsulation, where the class variables will be hidden from other classes, and can be accessed only through the methods of their current class.

- Encapsulation is also referred to as *data hiding*, which means to combine the data and the functions that manipulate the data into one place, keeping both safe from outside interference and misuse. Encapsulation allows the state of an object to be accessed and modified through behavior. It reduces the coupling of modules and increases the cohesion inside them.
- Encapsulation allows the programmer to protect the integrity of the data by creating restrictions that prevent invalid data from being input.
- The Setter and the Getter methods allow additional functionalities, such as validation and error handling to be added more easily in the future.

Knowledge Check

- What is Encapsulation in Java? Why is also referred to as *data hiding*?
- What are the important features of Encapsulation?
- How can you achieve Encapsulation in Java?

Lesson 2

Inheritance

One of the pillars of Object-Oriented Programming

Learning Objectives:

By the end of this lesson, learners will be able to:

- Understand the concepts of inheritance, superclass, and subclass.
- Describe the inheritance related topics such as Super keyword, Static Keyword, and Object Type Casting.
- Demonstrate the use of inheritance related topics
- Demonstrate how to call a constructor or method that is defined in a superclass.

Outline/Agenda

Topic 1a: Inheritance.

Topic 1b: Making a class inherit from another.

Topic 2: Method Overriding in Inheritance.

Topic 3: The Super Keyword.

Topic 4: The Static Keyword.

Topic 4a: The Static Variable.

Topic 4b: Static Method.

Topic 1a: Inheritance

- Inheritance is the process or ability by which one class can acquire the properties and behaviors of another class. The original class is known as the **parent or super** class, while the **new** class is known as the **child (subclass, derived class).**
  - The child class can also introduce/declare completely new variables, methods, and other attributes.
- In OOP, **Inheritance represents the IS-A relationship, which is also known as a parent-child relationship**.
  - Car is a ***subclass*** of Vehicle class.
  - Engine is a ***subclass*** of Part Class.
  - Rectangle is a ***subclass*** of Shape Class.
  - Checkingaccount is a ***subclass*** of Bankaccount Class.
  - Surgeon is a ***subclass*** of Doctor Class.
  - HumanBeing is a ***subclass*** of livingThing Class
- Java forces a class to have exactly **one parent** ("single inheritance").
- In OOP, **has a** relationships are expressed through composition.

Go to the **Wiki** document, for more details about inheritance

Topic 1b: Making a Class Inherit From Another

- Use the **`extends`** Keyword.
- CheckingAccount **`extends`** BankAccount.
- Car **`extends`** Vehicle.
- Engine **`extends`** Part.
- Rectangle **`extends`** Shape.
- Checkingaccount **`extends`** Bankaccount.
- HumanBeing **`extends`** livingthing.
  - HumanBeing class inherits all members of livingthing class.
  - HumanBeing can access all public and protected members of livingthing.
  - Non-default constructors are accessible, but not inherited.

Example 1 - Inheritance

Step 1: Create a class named ***livingThing***, and write the below code. This class will be our **parent** class.

```
public class livingThing {
// field and method of the parent class

   protected String name;
   public void eat() {
       System.out.println("I can eat");
   }
}
```

Step 2: Create a class named ***Humanbeing***, and write the below code. This class will be our **child** class.

```
public class Humanbeing extends livingThing {
     public void display() {
       System.out.println("My name is " + name);
   }
}
```

Step 3: Create a class named ***myMain***, and write the code below. This class will be our **main** class or entry point.

```
public class myMain {

   public static void main(String[] args) {
       System.out.println("Hello World!");

       // create an object of the subclass
       Humanbeing HObj = new Humanbeing();

       // access field of superclass
       HObj.name = "Mr Best";
       HObj.display();

/* call method of superclass by using object of subclass */
       HObj.eat();
   }
}
```

Output

```
Hello World!
My name is Mr Best.
I can eat.
```

Topic 2: Method **Overriding** in Inheritance

- The method in the **subclass** *overrides* the method in the ***superclass***. This concept is known as ***method overriding*** in Java.
- A Child class can replace the logic of a Parent class method.
  - But the **method name**, **return type**, and **parameters** will remain the same in the parent class method.
- @Override is an annotation that informs the compiler that a method is intended to override a method in a superclass. But It is not mandatory.
  - It is a best practice to use it because it gives you a safety net, just in case the developer changes a method name in a superclass.

Go to **Wiki** document, for more details about Method Overriding.

Example - Method **Overriding** in Inheritance

Step 1: Create a class named ***livingThing***, write a below code, this class will be parent class

```
public class livingThing {

// field and method of the parent class
   protected String name;
   public void eat() {
       System.out.println("I can eat");
   }
}
```

Step 2: Create a class named ***Humanbeing*** ,write a below code, this class will be our child class

```
public class Humanbeing extends livingThing {
      public void display() {
       System.out.println("My name is " + name);
   }
   // overriding the eat() method
   @Override // this is optional
   public void eat() {
       System.out.println("I enjoy eating Pizza with soft
drink");
   }

   // Declaring new method in subclass
   public void communication() {
       System.out.println("I can communicate by language");
   }}
```

Step 3: Create a class named ***myMain***, write a below code, this class will be out main class or entry point

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

       System.out.println("Hello World!");
       // create an object of the subclass
       Humanbeing HObj = new Humanbeing();

       // access field of superclass
       HObj.name = "Mr Best";
       HObj.display();

       HObj.eat();
       HObj.communication();
   }
}
```

Output

```
Hello World!
My name is Mr Best.
I enjoy eating Pizza with soft drink.
I can communicate by language.
```

Topic 3: The **Super** Keyword

- The super keyword in Java is a **reference variable** that is used to refer to the immediate superclass (parent) of current class (child).
- The ***Super*** keyword is used to ***invoke*** immediate:
  - parent class's ***methods*** in child class.
  - parent class's ***variables*** in child class.
  - parent class's ***constructors*** in child class.

➢ Syntax:

```
super() /*Calls the parent's constructor*/
super.fieldName /*Access a parent's field */
super.methodName() /*Access a parent's method */;
```

Go to **Wiki** document, for more details and explanation of Super Keyword

Important points:

1. Call to super() must be first statement in Derived(Student) Class constructor.
2. If a constructor does not explicitly invoke a superclass constructor, the Java compiler automatically inserts a call to the no-argument constructor of the superclass. If the superclass does not have a no-argument constructor, you will get a compile-time error.

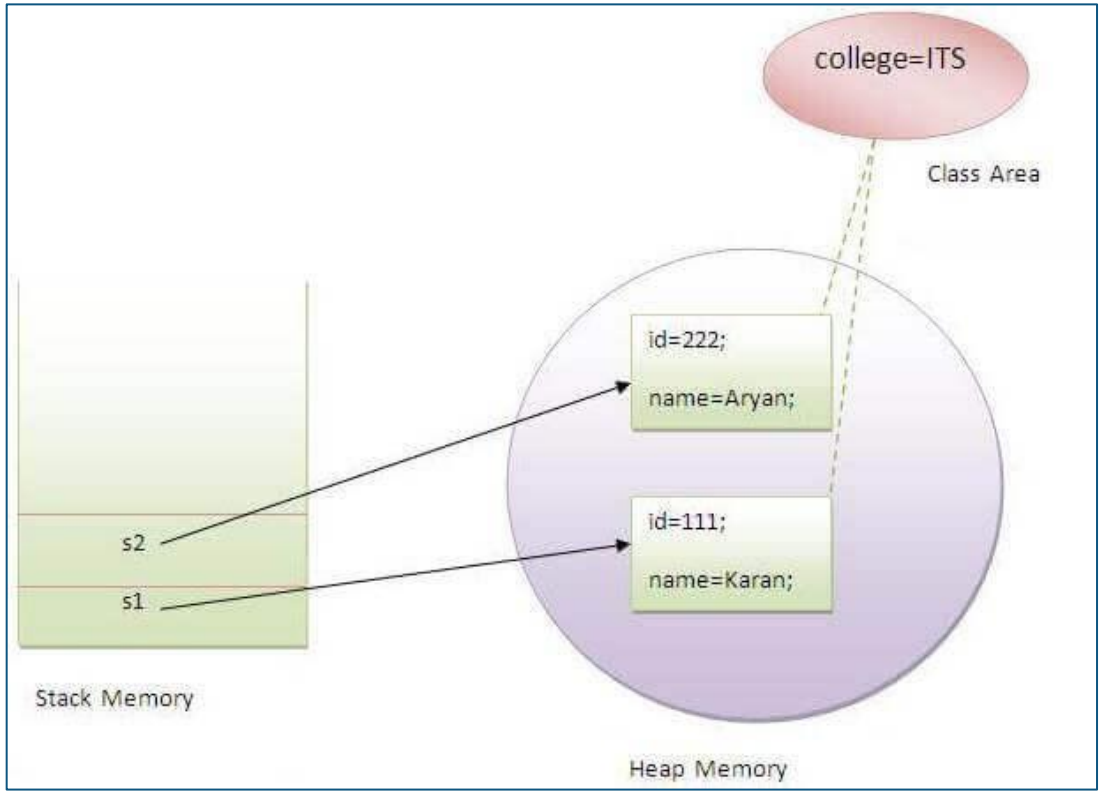
Topic 4: The **Static** Keyword

- What does the **Static** Keyword do?
  - Identifies a ***variable*** or ***method*** as **belonging to the class** rather than belonging to an instance of the class.
    - This means you can call the ***variable*** and ***method*** **directly from the class** by using **do(.)** notation
  - Allows that variable and method to consume less memory, as it will not be instantiated because a single copy of the **static variable** and **method** is created and shared among all the instances of the class.
  - Memory allocation for such variables only happens once when the class is loaded in the memory.
- Instantiated variable vs. Static variable.
  - Modifying an **instantiated** variable only changes it **for that instance.**
  - Modifying a **static** variable changes it **for all instances.**

Topic 4a: The **Static** Variable

The ***static*** variable can refer to the common property of all objects (which is not unique for each object). For example, the common property may be the company name of employees or the college name of students, etc.

The static variable gets memory only once in the class area during class loading. The advantage of static variable is that it makes your program memory efficient (i.e., it saves memory).



Example One: **Static** Variable


```
public class DemoOne {
    static int var1 = 20;
    static String var2 = "perscholas";
    int var3 = 100;
}
```

```
public class mymain {
    public static void main (String [ ] args){
        System.out.println(DemoOne.var1);
        System.out.println(DemoOne.var2);
          }
}
```

- `static` variables can be created at class level only.
- A `static` variable is allocated memory once.

A static variable can be accessed/invoke directly by the class name using dot(.) notation and doesn't need any object

Example Two: **Static** Variable


```
public class mymain {
   public static void main (String [ ] args){
        DemoOne obj1 = new DemoOne();
        DemoOne obj2 = new DemoOne();
        DemoOne obj3 = new DemoOne();
        DemoOne obj4 = new DemoOne();
        DemoOne obj5 = new DemoOne();
        /* only one copy of static variable will create and shared
         * among all the instances of class but 5 copies of class variable will be
create */
       System.out.println(" invoking static variable");
        obj1.var1 = 30;
        obj2.var1 = 40;
        obj3.var1 = 50;
        obj4.var1 = 60;
        obj5.var1 = 70;
        System.out.println(obj1.var1);
        System.out.println(obj2.var1);
        System.out.println(obj3.var1);
        System.out.println(obj4.var1);
        System.out.println(obj5.var1);

       System.out.println(" invoking non static variable");
        obj1.var3 = 200;
        obj2.var3 = 300;
        obj3.var3 = 400;
        obj4.var3 = 500;
        obj5.var3 = 600;
        System.out.println(obj1.var3);
        System.out.println(obj2.var3);
        System.out.println(obj3.var3);
        System.out.println(obj4.var3);
        System.out.println(obj5.var3);
   }
}
```

```
public class DemoOne {
    static int var1 = 20;
    static String var2 = "perscholas";
    int var3 = 100;
}
```

Output

```
 invoking static variable
70
70
70
70
70

 invoking non static variable
200
300
400
500
600
```

For static: Only one copy of static variable will create and shared among all the objects (obj1, obj2, obj3, obj4, obj5) of class

TEKsystems Own change

Topic 4b: **Static** Method

- A static method is **NOT** part of the ***objects(instance of a class);*** it creates but is part of a ***class definition.*** A static method is referenced by the class name and can be invoked without creating an object of class.
- They are accessed by the class name using a dot (.) notation, followed by the name of a method. They are declared with the keyword "static" when defining a method.

To learn more about the **static method**, visit the **Wiki** document.

Hands-On Lab

Complete **LAB - 303.5.2 - Basic Inheritance**. You can find the lab on Canvas under the **Guided Lab section**,

If you have any technical questions while performing the lab activity, ask your instructors for assistance.

Knowledge Check

- What is Inheritance in Java? Why do we need to use inheritance?
- How is Inheritance implemented/achieved in Java?
- Are static members inherited to subclass in Java?
- Can we override the static method in Java?

Practice Assignments - #**303.5.1**

These practice assignments will be administered through HackerRank. Click on the link below:

1. Java Inheritance I.
2. Java Inheritance II.
3. Java Method Overriding.
4. Java Method Overriding 2 (Super Keyword).

Use your office hours to complete this assignment. If you have any technical questions while in the lab, you can ask the instructor.

Summary

- Inheritance is one of the key features of object-oriented programming (OOP) that allows us to create a new class from an existing class. The new class is known as a subclass (child or derived class), and the existing class from where the child class is derived is known as a superclass (parent or base class).
- The extends keyword is used to perform inheritance in Java.
- The child object has an "is-a" relationship with the parent.
- The super keyword in Java is used in subclasses to access superclass members (attributes, constructors and methods), whereas the Static Keyword identifies what variables and methods should **not** be instantiated, saves memory every time a class is instantiated, and can be called directly from the Class.
- Overriding means that the logic of parent methods can be overridden by a child class.
- The static keyword can be applied to class members. This implies that the class members (static ones) belong to the class, not to instances of the class.

References

https://www.freecodecamp.org/news/java-getters-and-setters/

https://www.tutorialkart.com/java/encapsulation-in-java/