

Object Oriented Programming

Prepared by Dr. Yasser Abdelhamid

IT223
Object Oriented Programming

Lecture 07



OUTLINE

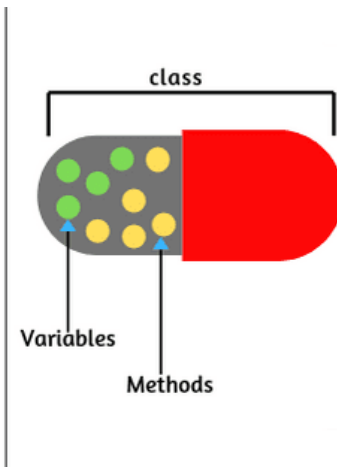
- Encapsulation
 - using private attributes and methods
 - getters and setters
 - this keyword
- Polymorphism
 - Purpose of polymorphism
 - Types of polymorphism
 - ad hoc polymorphism
 - Universal polymorphism

ENCAPSULATION

ENCAPSULATION

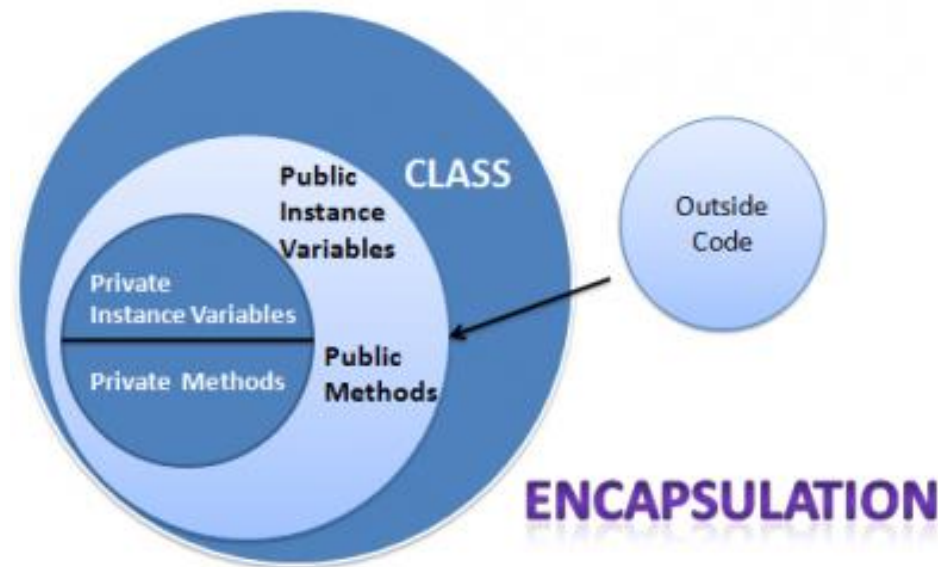
❖ **Encapsulation**, is a property of Object Oriented Programming that aims at making sure that "sensitive" data is hidden from users.

```
class  
{  
  
    data members  
    +  
    methods (behavior)  
  
}
```



ENCAPSULATION

- ❖ Encapsulation promotes the idea of **data hiding** and provides a way to **control access** to the internal state of an object.



ENCAPSULATION

- ❖ In Java, encapsulation is achieved by using access modifiers to restrict direct access to the fields (data) of a class and providing public methods (getters and setters) to access and manipulate the data.
- ❖ This ensures that the internal state of an object can only be modified through defined methods, maintaining data integrity and encapsulating the implementation details.

BENEFITS OF ENCAPSULATION

BENEFITS OF ENCAPSULATION

- ❖ Data Hiding: Encapsulation allows hiding the internal representation and implementation details of a class.
- ❖ The fields can be declared as private, preventing direct access from outside the class, and providing controlled access through public methods.
- ❖ This protects the data from being modified or accessed in an unintended manner.

BENEFITS OF ENCAPSULATION(CONT.)

- ❖ Access Control: By using access modifiers like private, protected, and public, encapsulation allows fine-grained control over the accessibility of class members.
- ❖ This helps in maintaining the desired level of security and abstraction, preventing unauthorized modifications or access to internal data.

BENEFITS OF ENCAPSULATION(CONT.)

- ❖ **Code Flexibility and Maintainability:**
Encapsulation provides a clear separation between the internal implementation of a class and its external interface.
- ❖ This allows modifications to the internal implementation without affecting the code that uses the class, enhancing code flexibility and making it easier to maintain and evolve the codebase.

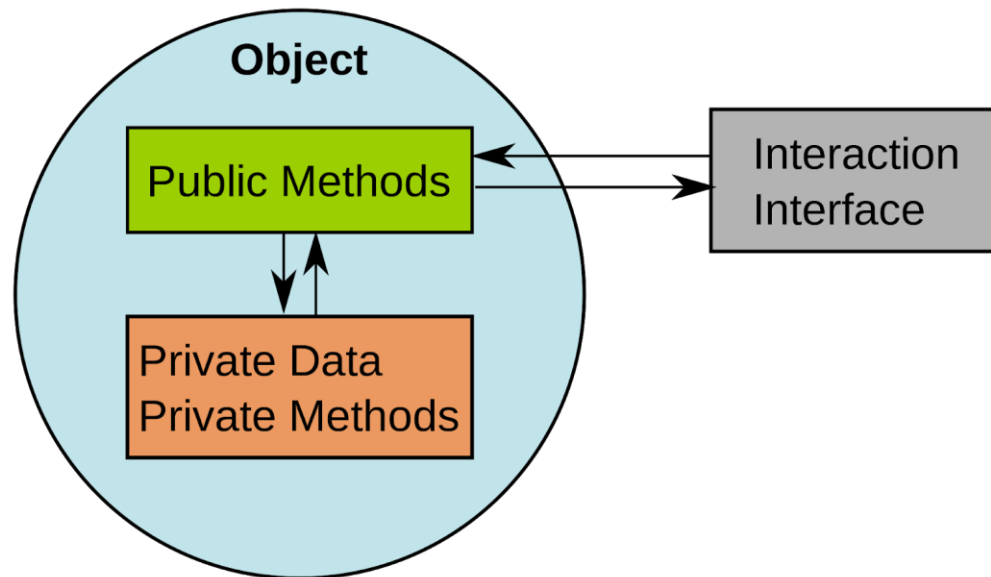
BENEFITS OF ENCAPSULATION(CONT.)

- ❖ Code Reusability: Encapsulation supports the creation of reusable code components by exposing a well-defined public interface while keeping the internal details hidden.
- ❖ Other parts of the codebase can use the class without being concerned about its internal implementation, promoting code reuse and modularity.

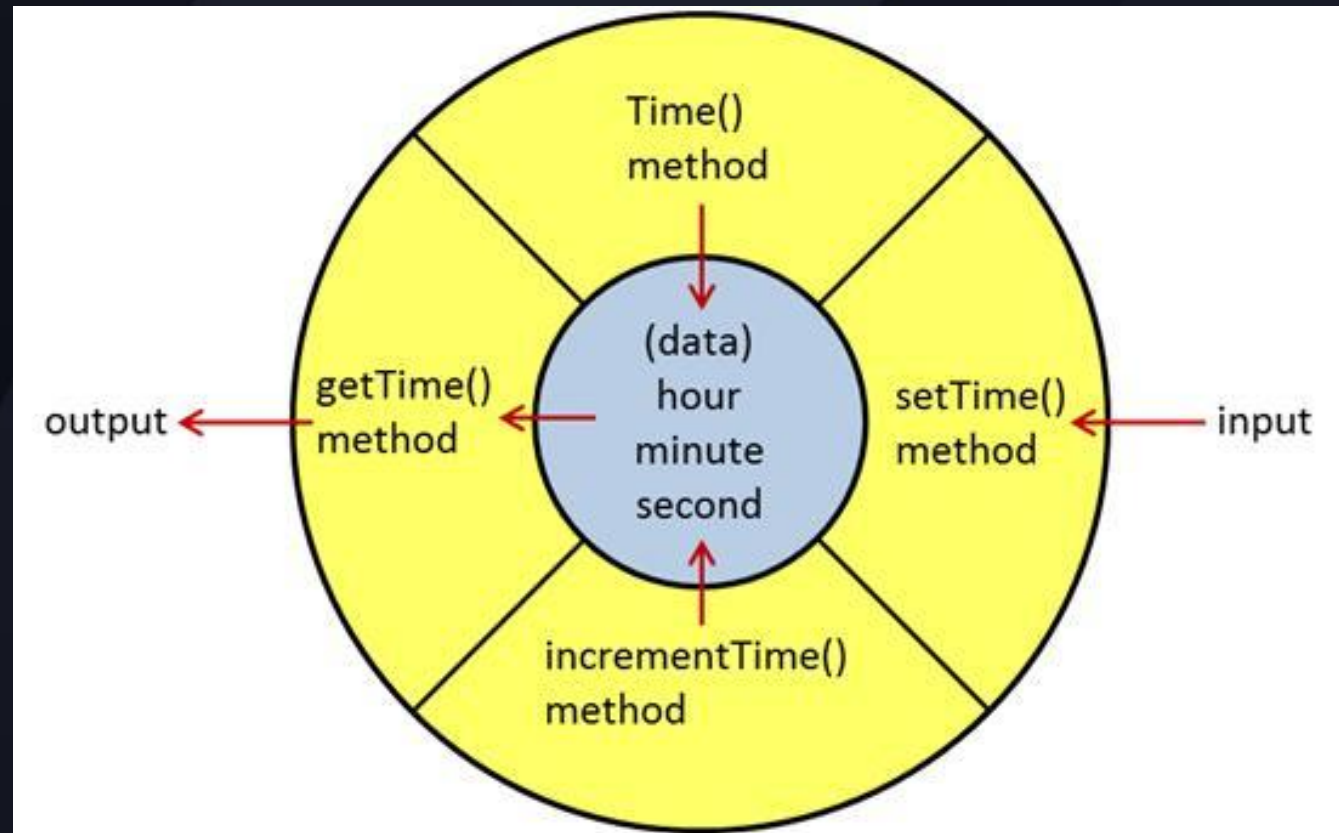
JAVA IMPLEMENTATION OF ENCAPSULATION

HOW TO IMPLEMENT ENCAPSULATION

- ❖ Declare class attributes and methods as **private**
- ❖ Provide **public get and set methods** to access and update the value of a private attribute, or call private methods.



EXAMPLE TIME CLASS



DETAILED EXAMPLE

❖ Declare the class:

- Start by declaring a class that will encapsulate the data and methods related to a specific entity or concept.

```
public class Person {  
    // class members and methods will be  
    defined here  
}
```

EXAMPLE

❖ Declare private fields:

- Declare the fields (variables) of the class as **private**. This makes them inaccessible from outside the class.

```
public class Person {  
    private String name;  
    private int age;  
    // other fields and methods will be defined  
    here  
}
```

EXAMPLE

- ❖ Provide public getter and setter methods:
 - Create **public** methods (**getters and setters**) to access and modify the private fields.
 - These methods provide **controlled access** to the class's data.

EXAMPLE

```
public class Person {  
    private String name;  
    private int age;  
    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;  
    }  
    // other methods will be defined here  
}
```

EXAMPLE

❖ Access the class through public methods:

- When using the class from other parts of the code, access its data and modify it using the public getter and setter methods.
- This ensures that the data is accessed and modified in a controlled manner.

EXAMPLE

```
public class Main {  
    public static void main(String[] args) {  
        Person person = new Person();  
        person.setName("John");  
        person.setAge(25);  
  
        System.out.println("Name: " + person.getName());  
        System.out.println("Age: " + person.getAge());  
    }  
}
```

```
public class Person {  
    private String name;  
    private int age;  
    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;  
    }  
    // other methods will be defined here  
}
```

TEST CLASS

```
class Test {  
    public static void main(String[] args) {  
        Person p1 = new Person();  
        p1.setName("Sara");  
        // p1.name = "Mohamed"; Can I do this?  
        System.out.println(p1.getName());  
    }  
}
```

```
public class Person {  
    private String name;  
    private int age;  
    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;  
    }  
    // other methods will be defined here  
}
```

QUICK QUESTIONS

- ❖ How can I make an attribute READ ONLY?
- ❖ How can I make an attribute WRITE ONLY?
- ❖ What are the advantages of encapsulation?
- ❖ Can I use a class in different package without using the **import** keyword?

ACTIVITY

```

/*****

```

- * Complete this class to do the following:
 - * Implement getter and setter for the age attribute
 - * Age should be within the range of 20 to 40 inclusive
 - * Implement a private method to check name as follows:
 - * Name should not be empty
 - * Name should include only English characters
 - * Length of Name should not exceed 20 characters.
 - * Can we implement one method to get person data
 - * Can we implement one method to set person data
 - * Add phone number and email to the Person class
 - * Add birth date that contains Day, Month, and Year
 - * Validate phone number (should be 11 digits starting with 010, 012, or 015)
 - * Validate email (should be in the pattern username@subdomain.domain)
 - * Implement birth date as a class (Do not use Java Date class)
 - * Implement the required methods to set and get day, month, and year
 - * Implement the required methods to validate day, month and year
- ```

*/

```

# POLYMORPHISM

# POLYMORPHISM

- ❖ In programming, **polymorphism** is the ability of an entity (e.g. variable, class, method, object, code, parameter, etc.) to have different meanings in different contexts.
- ❖ The entity that has different meanings is known as a **polymorphic entity**.

# PURPOSE OF POLYMORPHISM

- ❖ The purpose of polymorphism is writing code that is:
  - Reusable and
  - Maintainable
- ❖ This can be achieved by writing generic code that works for many types of data (or ideally all types).

# TYPES OF POLYMORPHISM

## ❖ Ad hoc polymorphism (apparent polymorphism)

- If the types for which a piece of code works are **finite** and all those types must be known when the code is written.

## ❖ Universal polymorphism

- If a piece of code is written in such a way that it works for **infinite** number of types (will also work for new types not known at the time the code is written)

# AD HOC POLYMORPHISM

- ❖ Overloading is an ad hoc polymorphism.
- ❖ Method overloading gives only sharing of the method name.
- ❖ They may perform entirely different things and share the same name.

# EXAMPLE

```
public class MathUtil {
 public static int process1(int n1, int n2) {
 }
 public static double process1 (double n1, int n2) {
 }
 public static double process1 (int n1, double n2) {

 public static int process1 (int[] num) {
 }
}
```

```
int p1 = MathUtil.process1(10, 23); // Uses process1(int, int)
double p2 = MathUtil.process1(10.34, 2); // Uses process1(double, int)
double p3 = MathUtil.process1(3,10.34); // Uses process1(int,double)
```

# COERCION POLYMORPHISM

❖ Coercion polymorphism occurs when a type is **implicitly** converted (coerced) to another type **automatically** even if it was not intended explicitly.

# COERCION POLYMORPHISM

❖ Suppose  $m$  is a method that declares a parameter of type  $T$ . If  $S$  is a type that can be implicitly converted to  $T$ , the method  $m$  is said to be **polymorphic with respect to  $S$  and  $T$** .

Example:

```
int num = 707;
```

```
double d1 = (double)num; // Explicit conversion
```

```
double d2 = num; // Implicit conversion (coercion)
```

# EXAMPLE

```
double square(double num) {
 return num * num;
}
```

```
// calling square with double argument.
```

```
double d1 = 20.23;
```

```
double result = square(d1);
```

```
// calling square with integer argument.
```

```
int k = 20;
```

```
double result = square(k);
```

# INCLUSION POLYMORPHISM

- ❖ Inclusion is a universal polymorphism.
- ❖ Inclusion polymorphism occurs when a piece of code that is written using a class, works for all its subclasses.
- ❖ It is also known as subtype (or subclass) polymorphism because it is achieved using subtyping or sub-classing.

# INCLUSION POLYMORPHISM

- ❖ This is the most common type of polymorphism supported by object-oriented programming languages.
- ❖ This type of polymorphism is possible based on the subclassing rule that a value that belongs to a subclass also belongs to the super class.

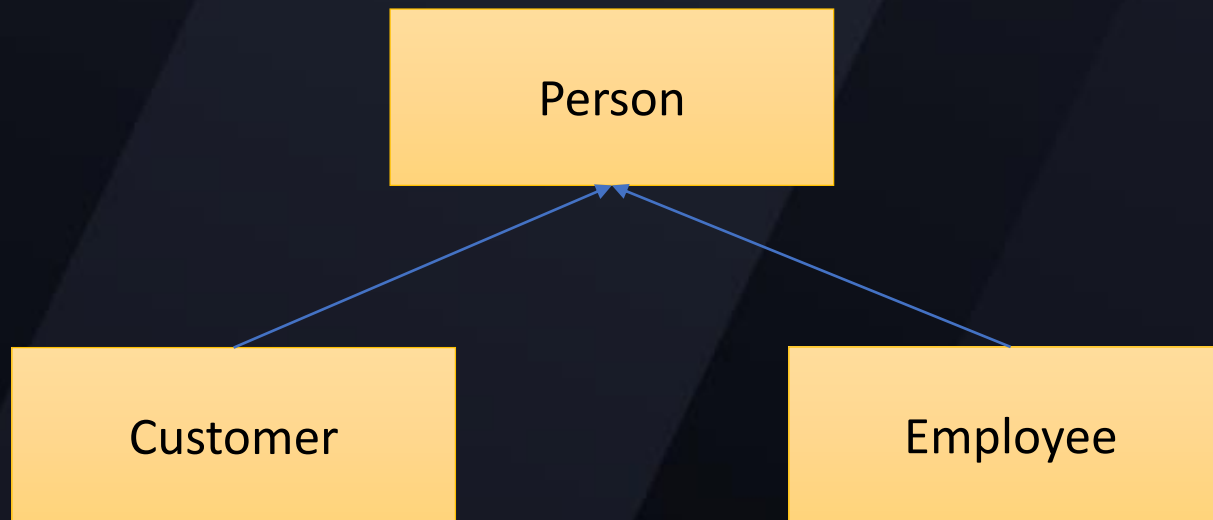
## EXAMPLE

- ❖ Suppose T is a class and S1, S2, S3... are subclasses of T.
- ❖ A value that belongs to S1, S2, S3... also belongs to T.
- ❖ This subtyping rule makes us write code as follows:

## EXAMPLE

```
T t; // t is an instance of the superclass T
S1 s1; // s1 is an instance of the subclass S1
S2 s2; // s2 is an instance of the subclass S2
...
// A value of type s1 can be assigned to variable of type T
t = s1;
// A value of type s2 can be assigned to variable of type T
t = s2;
```

# EXAMPLE



# EXAMPLE

```
void processDetails(Person p) {
 /* Write code using the parameter p, which is of type Person.
 The same code will work if an object of any of the subclass of Person is passed to this
 method.
 */
}
```

```
Person p1 = new Person();
Employee e1 = new Employee(); // Employee is a subclass of Person
Customer c1 = new Customer(); // Customer is a subclass of Person
processDetails(p1); // Use Person type
processDetails(e1); // Use Employee type, which is a subclass of Person
processDetails(c1); // Use Customer type, which is a subclass of Person
```

# PARAMETRIC POLYMORPHISM

- ❖ In parametric polymorphism, a piece of code is written in such a way that it works on **any class**.
- ❖ In inclusion polymorphism, code is written for one class and it works for **all of its subclasses**.
- ❖ However, in parametric polymorphism, the same code works **for all classes, which are not necessarily related**.

# PARAMETRIC POLYMORPHISM

❖ Parametric polymorphism is achieved by using a **type variable** when writing the code, rather than using any specific type.

# EXAMPLE

// This example is using collections, to be covered later in this course.

```
List<String> sList = new ArrayList<String>();
sList.add("string 1");
sList.add("string 2");
String s2 = sList.get(1);
// Use List for Integer type
List<Integer> iList = new ArrayList<Integer>();
iList.add(10);
iList.add(20);
int i2 = iList.get(1);
```

# SUMMARY

- Encapsulation
  - using private attributes and methods
  - getters and setters
  - this keyword
- Polymorphism
  - Purpose of polymorphism
  - Types of polymorphism
    - ad hoc polymorphism
    - Universal polymorphism



**Thank You**

[www.midocean.ae](http://www.midocean.ae)

