VERSION 1.6

14 March 2025

OBJECT ORIENTED PROGRAMMING

MODULE 3 – CONSTRUCTOR, ENCAPSULATION, INHERITANCE, OVERRIDING, SUPER KEYWORDS

COMPILED BY:

WIRA YUDHA AJI PRATAMA

KEN ARYO BIMANTORO

AUDITED BY:

Ir. Galih Wasis Wicaksono, S.Kom, M.Cs.

PRESENTED BY: TIM LAB. IT

UNIVERSITAS MUHAMMADIYAH MALANG

INTRODUCTION

OBJECTIVE

1. Students understand the concepts of constructors, encapsulation, inheritance, overriding, and the use of the super keyword in Java.
2. Students understand how these concepts are used to enhance modularity and efficiency in object-oriented programming.
3. Students understand how to implement these OOP concepts in more complex Java projects.

MODULE TARGET

1. Students can create a program that applies constructors, encapsulation, inheritance, and overriding in Java.
2. Students can use the super keyword to access constructors or methods from the superclass.

PREPARATION

1. Device (Laptop/Computer)
2. IDE (IntelliJ)
3. Internet

KEYWORDS

Encapsulation, Inheritance, Overriding, Super Keyword

TABLE OF CONTENTS

CONSTRUCTOR

THEORY

A constructor is a method that initializes an object by assigning initial values when the object is created. When the program runs, the constructor automatically provides initial values when an instance of an object is created.

MATERIALS

When working with constructors, there are several important points to consider:

1. The constructor name must be the same as the class name.
2. No return type is specified in the constructor signature.
3. There is no return statement inside the constructor body.

Example of creating a Car class and using a constructor:

```
public class Car {
    String name;
    int speed;

    Car(){
        System.out.println("Constructor method executed");
    }
}
```

In the example above, the constructor is written as follows:

```
Car(){
    System.out.println("Constructor method executed");
}
```

Let's try to create a new object of the Car class in the Main class:

```
Car car = new Car();
```

So now the code in our Main class looks like this:

```
public class Main {
    public static void main(String[] args){
      Car car = new Car();
    }
}
```

When run it will display the following output:

```
Constructor method executed


Process finished with exit code 0
```

The constructor method will be executed when a new instance of an object is created, even without explicitly calling the method. There are two ways to write a constructor in a class:

1. Writing a method with the same name in a class

```
class Car {
    String name;
    int speed;

    Car() {
        System.out.println("Constructor method executed");
    }
}
```

2. Using the public modifier

```java
public class Car {
    String name;
    int speed;

    public Car() {
        System.out.println("Constructor method executed");
    }
}
```

Constructor with Parameters

After knowing what a constructor is, we will wonder what the function of the constructor itself is? The answer is with an example case, for example we have 5 car data with each car data having a different name and speed and want to create a car object one by one. Maybe the code we create will be like this:

- in the Carclass

```java
public class Car {
    String name;
    int speed;

    public Car(){
        System.out.println("Constructor method executed");
    }
}
```

- in the Main class

```
public class Main{
    public static void main(String[] args){
        Car car1 = new Car();
        Car car2 = new Car();
        Car car3 = new Car();
        Car car4 = new Car();
        Car car5 = new Car();

        car1.name = "Toyota Avanza";
        car1.speed = 160;
        car2.name = "Suzuki SX4 2007";
        car2.speed = 170;
        car3.name = "Sedan car";
        car3.speed = 210;
        car4.name = "BMW i8";
        car4.speed = 190;
        car5.name = "Bugatti Chiron";
        car5.speed = 240;
    }
}
```

in the Main class code it can be seen that we have to input one by one and take up a lot of space just to input one data. Therefore, if we want to initialize data directly when a class instance is created, then we can use a parameter by adding a parameter to the Car class constructor and using arguments when creating an instance of the Car class object. The code will look like this.

- in the Car class

```
class Car {
    String name;
    int speed;

    public Car(String name, int speed){
        this.name = name;
        this.speed = speed;
        System.out.println("Constructor method executed");
    }
}
```

- in the Main class

```
public class Main{
    public static void main(String[] args){
        Car car1 = new Car("Toyota Avanza", 160);
        Car car2 = new Car("Suzuki SX4 2007", 170);
        Car car3 = new Car("Sedan car", 210);
        Car car4 = new Car("BMW i8", 190);
        Car car5 = new Car("Bugatti Chiron", 240);
    }
}
```

In the Car class code above, we add the name and speed parameters to the constructor. So when creating an object instance of the Car class, we have to add arguments like this:

```
Car car5 = new Car("Bugatti Chiron", 240);
```

The difference is very visible when we do not use a constructor and when using a constructor, the code will be simpler and easier to read when using a constructor.

If we do not create a constructor in a class, then automatically and unknowingly the class has actually created its own constructor but the constructor body is empty.

PRACTICE

Let's apply this Constructor technique to our previous project! Please open the project in the previous module practice. Is it still there? If not, please recreate it 🙂

```java
import java.util.Date;

public class Main {
    public static void main(String[] args) {
        Farmer farmer1 = new Farmer();
        Farmer farmer2 = new Farmer();
        Plant plant1 = new Plant();
        Plant plant2 = new Plant();

        farmer1.name = "Crazy Dave";
        farmer2.name = "Sober Dave";

        plant1.name = "Sunflower";
        plant2.name = "Mushroom";

        farmer1.favourite = plant1.name;
        farmer2.favourite = plant2.name;

        System.out.println("Hello, World!");
        System.out.println("Current date and time: " + new Date());

        farmer1.talk();
        farmer2.talk();
    }
}
```

We can make some changes to the Main class to make it shorter and more efficient. How? Please go to the Farmer and Plant Class to create a constructor that we will use in the Main Class.

```java
public class Farmer {
    2 usages
    String name;
    2 usages
    String favourite;

    1 usage
    public Farmer(String name, String favourite) {
        this.name = name;
        this.favourite = favourite;
    }
    1 usage
    void talk() {
        System.out.println("Hi! My name is: " + name + ". My favourite plant is: " + favourite);
    }
}
```

This is Constructor

```java
public class Plant {
    1 usage
    String name;
    1 usage
    public Plant(String name){
        this.name = name;
    }
}
```

This is Constructor

Very cool, right? You can also change the parameters using Objects. Please try it! If you have any difficulties, you can ask the assistant.

```java
import java.util.Date;

public class Main {
    public static void main(String[] args) {
        Plant plant1 = new Plant( name: "Sunflower");
        Plant plant2 = new Plant( name: "Mushroom");
        Farmer farmer1 = new Farmer( name: "Crazy Dave", plant1.name);
        Farmer farmer2 = new Farmer( name: "Sober Dave", plant2.name);

        System.out.println("Hello, World!");
        System.out.println("Current date and time: " + new Date());

        farmer1.talk();
        farmer2.talk();
    }
}
```

Very cool, right? You can also change the parameters using Objects. Please try it! If you have any difficulties, you can ask the assistant.

TIPS

Additional material about Constructors in Java:

Video1

VIdeo2

Materi

ENCAPSULATION

THEORY

Encapsulation is a wrapper, encapsulation in object oriented means wrapping a class and keeping everything in the class, both methods and attributes, so that it cannot be accessed by other classes. To maintain this in Encapsulation, the name Modifier Access Rights is known, which consists of Private. Public and Protected. But keep in mind, modifiers can not only be given to attributes and methods. But can also be given to interfaces, enums, and the class itself.

MATERIALS

The function of the access modifier in Java is to limit the scope of a class, constructor, variable, method, or other data member contained in a Java program.

Modifier	Class	Package	Subclass	World
public	True	True	True	True
protected	True	True	True	False
no modifier	True	True	False	False
private	True	False	False	False

Pay attention to the following code:

```java
public class Car {
    private String name;

    public int speed;

    protected boolean isDrive;

    public Car(){
        System.out.println("Constructor method executed");
    }
}
```

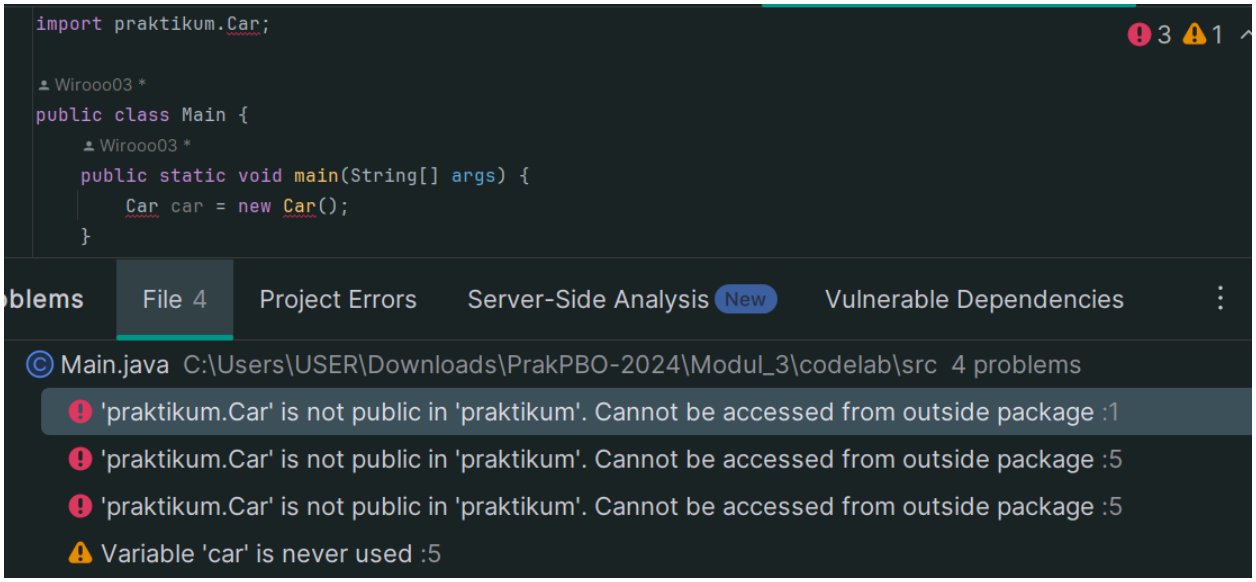
The code marked above is a modifier. Modifiers will be used to determine attribute, method, and class access.

Public

Granting access rights to attributes or methods so that they can be accessed by anyone (property or other classes outside the class concerned), meaning that methods or attributes in class A can be accessed by anyone, be it class A, class B and so on. Example:

```java
package praktikum;

class Car {
    private String name;

    public void changeName(String newName) {
        this.name = newName;
    }
}
```

In the Car class above, there is a name attribute and a changeName() method. We give both of these a public modifier, meaning that both can be accessed from anywhere. However, we do not give the Car class a modifier. So what will happen is that the class cannot be imported or accessed from outside the package.



The Car class is in the practicum package, then we try to access it from the main package, then an error will occur as shown in the image above. The way to fix this is to add the public modifier to the Car class. Then the code in the Car class will be like this:

```
package praktikum;

public class Car {
    private String name;

    public void setName(String newName) {
        this.name = newName;
    }
}
```

Then the error will disappear and the code can be run normally.

Protected

Granting access rights to the class itself and its derived classes (inheritance), meaning that anything in class A can only be accessed by class A itself and the class that extends class A. However, it must be understood that other classes in the same package as class A are able to access protected data types, while those that are not able to access them are classes that are outside the class A package. To be able to access it, classes outside the class A package must extend class A. Specifically, the protected modifier can only be used on attributes and methods, it cannot be given to classes, enums, and interfaces.

For example, the Car class has a derivative, namely the Honda class:

```java
package praktikum;

class Car {
    protected String name;

    public void changeName(String newName) {
        this.name = newName;
    }
}
```

In the Car class code above, we give the protected modifier to the name attribute.

```java
package praktikum;

public class Honda extends Sedan {
    public void drive() {
        name = "Civic";
        System.out.println("Driving a Honda car " + name);
    }
}
```

So when we try to access the Honda class which is a derived class, this will not cause an error. Or if we want to access from a class that is in the same package as the Car class, it can also be done even though it is not a derived class of the Car class.

However, if we try to access the name attribute from a different package, an error will occur because we have given the protected modifier to the name attribute.

```java
package Main;
import Praktikum.Car;

public class Main {
    public static void main(String[] args) throws Exception {
        Car car = new Car();
        // Use the setter method to avoid the error
        car.setName("Civic");
    }
}
```

Private

Granting access rights only to the class itself, meaning anything in class A, be it any method or attribute, can only be accessed by class A, other classes cannot access it. Private modifiers also cannot be given to classes, enums, and interfaces. Private modifiers are only given to class attributes and methods. Example:

```java
package praktikum;

public class Car {
    private String name;

    public void setName(String newName) {
        this.name = newName;
    }

    public String getName() {
        return name;
    }
}
```

In the example above, we give a private modifier to the name attribute and a public modifier to the setName() and getName() methods. If we try to directly access the name attribute like this:

```java
Car car = new Car();
car.name = "Civic"; // <= this will cause an error
```

So how do you access an attribute or method that is given a private modifier from outside the class? The answer is to use the getter and setter methods. Because, this method will always be given a public modifier. The setter and getter methods will be studied after this.

Setter and Getter Methods

Setter is a method when we enter a value into a variable/attribute/object, while Getter is a method when we take a value from a variable/attribute/object.

How to implement setters and getters in the Car class, first create several attributes with private modifiers. After that, create a regular method, for the setter, please start with the word set and for the getter, please start with get so that it is easy to know which code is included in the setter and getter. For the setter and getter method modifiers, always provide a public modifier, because this method will be accessed by other classes. For example:

```java
package praktikum;

public class Car {
    private String name;
    private String color;
    private int maxSpeed;
    private boolean isModified;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public String getColor() {
        return color;
    }

    public void setMaxSpeed(int maxSpeed) {
        this.maxSpeed = maxSpeed;
    }

    public int getMaxSpeed() {
        return maxSpeed;
    }

    public void setIsModified(boolean isModified) {
        this.isModified = isModified;
    }

    public boolean isModified() {
        return isModified;
    }
}
```

The main difference between the setter and getter methods lies in the return value, parameters, and contents of the method. The setter method does not have a void return value (empty), because its job is only to fill data into the attribute. While the getter method has a return value according to the type of data to be taken.

```java
// ini method setter
public void setName(String name){
    this.name = name;
}

// ini method getter
public String getName(){
    return name;
}
```

Sometimes we wonder, is it acceptable to use Indonesian for method prefixes? For example, using 'isi' for a setter and 'dapatkan' for a getter. While it is technically allowed, it is highly discouraged. This is because if you work in a team, especially with members from different countries who predominantly use English, they may find it difficult to understand the code.

After we create the setter and getter methods, we can access or use them like regular methods. Example:

```java
package main;
import praktikum.Car;

public class Main {
    public static void main(String[] args) throws Exception {
        Car car = new Car();

        // Using setter method
        car.setName("Civic");
        car.setMaxSpeed(180);

        // Using getter method
        System.out.println("Name: " + car.getName() + " " + "Max Speed: " + car.getMaxSpeed());
    }
}
```

Output:

```
Nama : Civic
Max Speed : 180

Process finished with exit code 0
```

1. Some reasons why you should use Setters and Getters:
2. To improve data security

3. To make it easier to control attributes and methods
4. We can make classes read-only and write-only
5. Programmers can replace part of the code without having to worry about impacting other code

PRACTICE

Let's apply this Encapsulation concept in our practical project! Please open the project file. Then go to the Farmer and Plant Classes, then make all variables private like this:

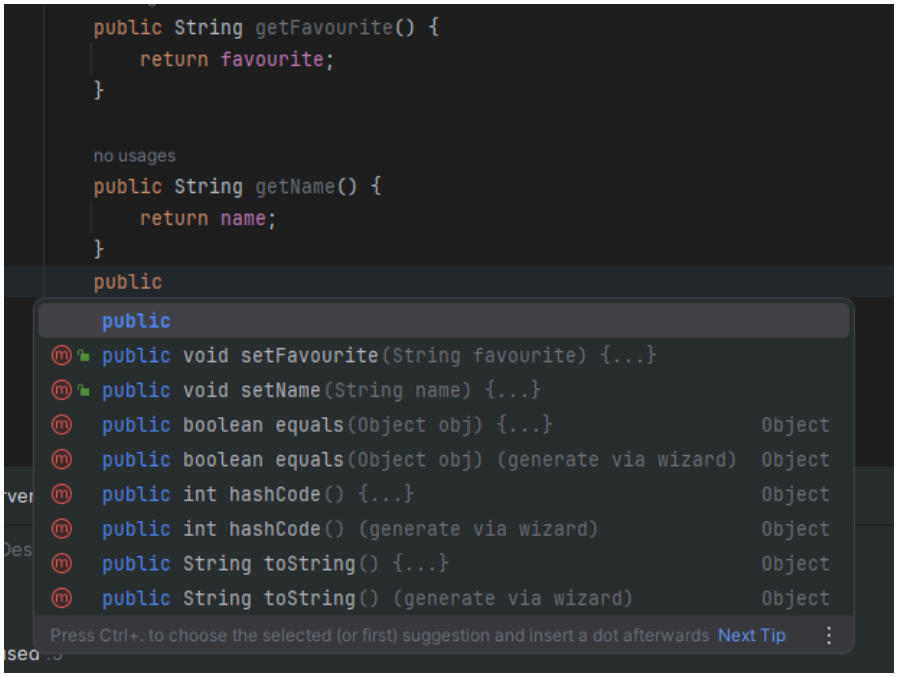
```
public class Plant {
    1 usage  2 related problems
    private String name;
    2 usages
    public Plant(String name){
        this.name = name;
    }
}
```

Wait, there is an error. What do you think happened? Ah right, in the main Class, we tried to call the 'name' variable in the Plant Class which is private. Then how do we get the 'name' variable? We will create Getter and Setter methods as follows:

```
public class Plant {
    2 usages
    private String name;
    2 usages
    public String getName(){
        return name;
    }
    no usages
    public String setName(String name){
        return name;
    }
    2 usages
    public Plant(String name){
        this.name = name;
    }
}
```

```java
public class Farmer {
    4 usages
    private String name;
    4 usages
    private String favourite;

    no usages
    public String getFavourite() {
        return favourite;
    }
    no usages
    public String getName() {
        return name;
    }
    no usages
    public void setFavourite(String favourite) {
        this.favourite = favourite;
    }
    no usages
    public void setName(String name) {
        this.name = name;
    }

    2 usages
    public Farmer(String name, String favourite){
        this.name = name;
        this.favourite = favourite;
    }
    2 usages
```

You can use the auto-complete feature provided by IntelliJ if you are too lazy to type.



Then in the Main Class, you change the Farmer class constructor to be like this:

```
import java.util.Date;

public class Main {
    public static void main(String[] args) {
        Plant plant1 = new Plant( name: "Sunflower");
        Plant plant2 = new Plant( name: "Mushroom");
        Farmer farmer1 = new Farmer( name: "Crazy Dave", plant1.getName());
        Farmer farmer2 = new Farmer( name: "Sober Dave", plant2.getName());

        System.out.println("Hello, World!");
        System.out.println("Current date and time: " + new Date());

        farmer1.talk();
        farmer2.talk();
    }
}
```

TIPS

Java encapsulation:

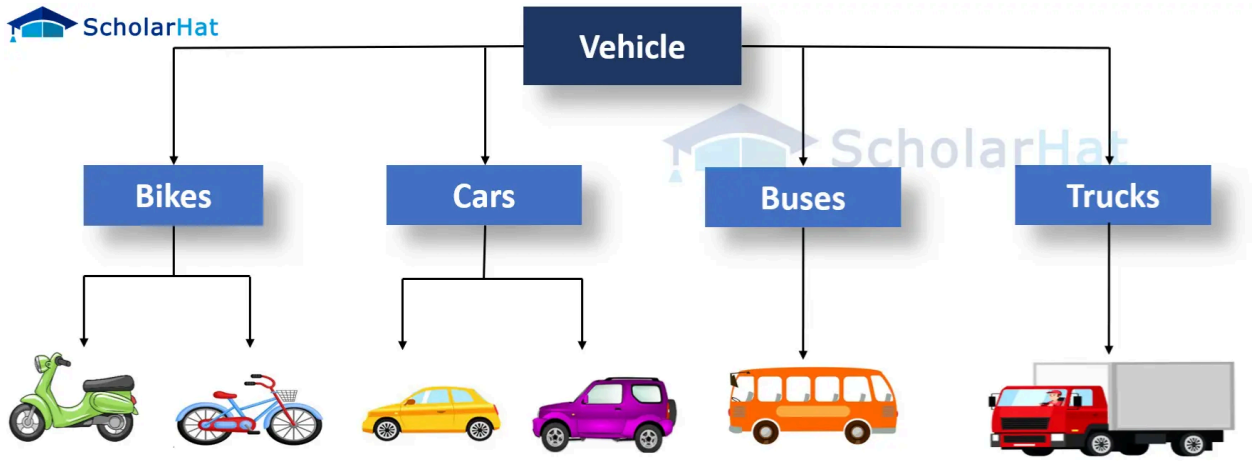
Video

Basic getters and setters:

Video

INHERITANCE

THEORY



Inheritance is a programming principle where a class has the ability to inherit its properties and methods to another class. The concept of inheritance is used to utilize the 'code reuse' feature to avoid duplication of program code.

With inheritance, a structure or hierarchy of classes is formed in the program code. The class that provides inheritance is called the parent class, super class, or base class. While the class that receives inheritance is called the child class, sub class, derived class, or heir class.

Not all properties and methods from the parent class will be inherited. Properties and methods with private access rights will not be inherited to the child class. Only properties and methods with protected and public access rights can be accessed from the child class.

A class that has a derivative is called the parent class or base class. While the derived class itself is often referred to as a subclass or child class. A subclass can inherit everything owned by the parent class.

Since a subclass can inherit everything that its parent class has, the members of a subclass consist of what it has and also what it inherits from its parent class. Overall, it can be said that a subclass simply extends its parent class.

MATERIALS

The reason why we have to use inheritance, for example in a car, we will create more specific car classes. Then we create code for each class like this:

File: Honda.java

```java
package praktikum;

public class Honda {
    public String name;

    public void moveForward() {
        System.out.println("The car " + name + " is moving forward");
    }

    public void turn() {
        System.out.println("The car is turning");
    }
}
```

File: Toyota.java

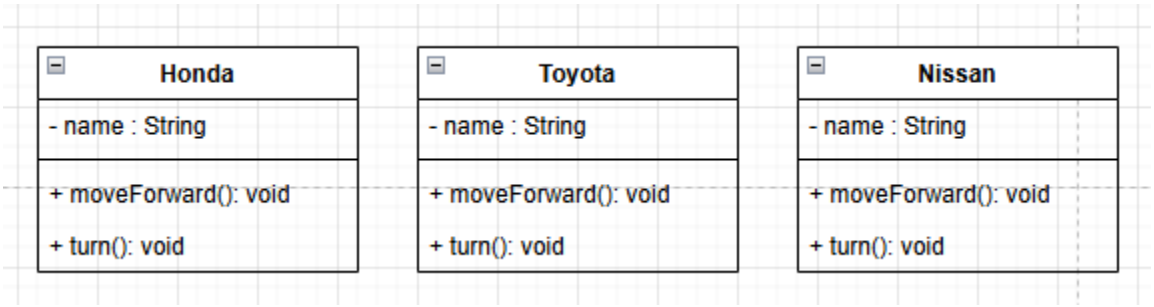
```java
package praktikum;

public class Toyota {
    public String name;

    public void moveForward() {
        System.out.println("The car " + name + " is moving forward");
    }

    public void turn() {
        System.out.println("The car is turning");
    }
}
```

File: Nissan.java

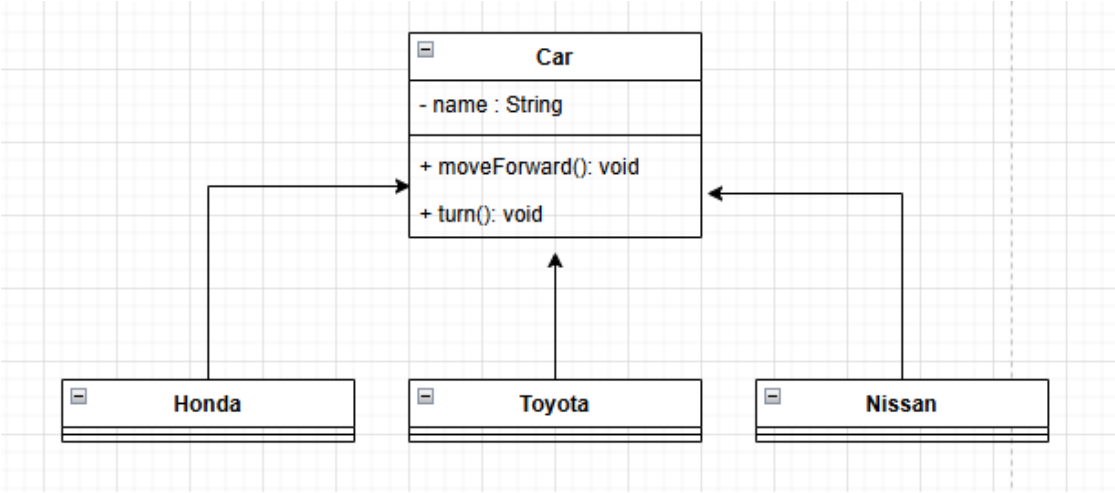
```java
package praktikum;

public class Nissan {
    public String name;

    public void moveForward() {
        System.out.println("The car " + name + " is moving forward");
    }

    public void turn() {
        System.out.println("The car is turning");
    }
}
```

From the three code classes above, is it permissible to write code like that? Yes, it is permissible. But it is not effective, because we write the same code over and over again.

Instead of repeating code, the solution that can be used is to use inheritance. Let's look at the same attributes and methods:



In the diagram above, it can be seen that the name attribute, the moveForward() and turn() methods have similarities in the Honda, Toyota, and Nissan classes. After we use inheritance by uniting attributes and methods that work the same way into one class, namely the Car class, the diagram will look like this:



The Car class is the parent class that has child classes, namely Honda, Toyota, and Nissan. Whatever attributes are in the parent class will also be owned by the child class. The Car class code looks like this:

```java
package praktikum;

public class Car {
    private String name;

    public void moveForward() {
        System.out.println("Car " + name + " is moving forward");
    }

    public void turn() {
        System.out.println("Car is turning");
    }
}
```

In the child class, we must use the extends keyword to state that the class is a child class of the Car class.

File: Honda.java

```java
package praktikum;
public class Honda extends Car {

}
```

File: Toyota.java

```java
package praktikum;
public class Toyota extends Car {

}
```

File: Nissan.java

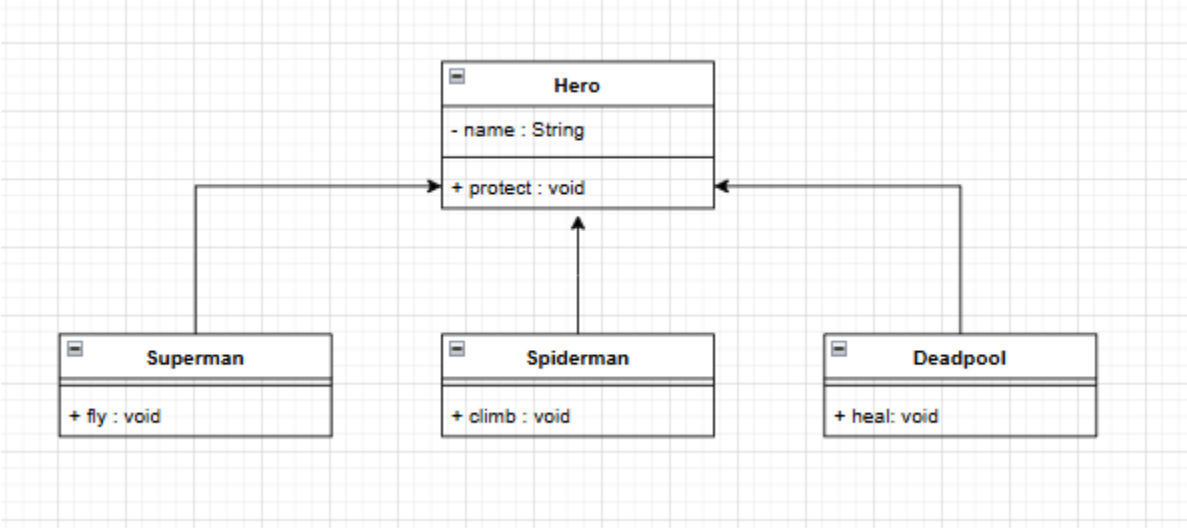
```java
package praktikum;
public class Nissan extends Car {

}
```

It can be seen in the Honda, Toyota, Nissan classes, the body class is empty, in the child class we can leave the body content empty or fill it as needed. For how to create objects from each class, we can make it like this:

File: Main.java

```java
Car car = new Car();
Honda honda = new Honda();
Toyota toyota = new Toyota();
Nissan nissan = new Nissan();
```

Another example case is the Hero class, for example the Hero class which has child classes in the form of Superman, Spiderman, Deadpool. The three child classes have the same name Hero, protecting the community. But each child class has a difference, namely Superman can fly, Spiderman can climb cliffs, Deadpool can heal himself. The diagram of the parent class and child class can be like this:



For the code for each class it will be like this:

File: Hero.java

```
public class Hero {
    private String name;

    public void protect(){
        System.out.println(name + " is protecting the community");
    }

    public void setName(String name){
        this.name = name;
    }

    public String getName(){
        return name;
    }
}
```

File: Superman.java

```
public class Superman extends Hero{
    public void fly(){
        System.out.println(getName() + " is flying...");
    }
}
```

File: Spiderman.java

```
public class Spiderman extends Hero{
    public void climb(){
        System.out.println(getName() + " is climbing cliffs");
    }
}
```

File: Deadpool.java

```
public class Deadpool extends Hero{
    public void heal(){
        System.out.println(getName() + " is healing themselves");
    }
}
```

To use it in the Main.java class file like this:

```
public class Main {
    public static void main(String[] args) {
        // pembuatan object
        Spiderman spiderman = new Spiderman();
        Superman superman = new Superman();
        Deadpool deadpool = new Deadpool();

        // memanggil method parent class melalui child class
        spiderman.setName("Tobey maguaire");
        superman.setName("Cristopher");
        deadpool.setName("Ryan Reynolds");

        // memanggil method yang ada di child class
        spiderman.memanjat();
        superman.terbang();
        deadpool.menyembuhkan();

        // Jika butuh object lain bisa membuat lagi
        Spiderman spiderman2 = new Spiderman();
    }
}
```

The call to the setName method is chosen by the parent class owned by the Hero class, it can be called through the child class because each class Spiderman, Superman, Deadpool is a derivative or child of the Hero class. While the climb() method is only owned by

the Spiderman class, then the Superman and Deadpool classes cannot call the climb() class because they do not have the class, also applies to the fly() method in the Superman class and the heal() method in the Deadpool class.

PRACTICE

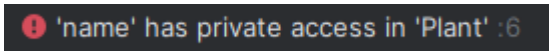
Let's practice the Inheritance technique we just learned in our program! Please create two new classes with the following specifications:

```
class Flower extends Plant {
    no usages
    public Flower(String name) {
        super(name);
    }
    no usages
    public void talk() {
        System.out.println("Hi! I'm " + name + "! And I'm a Flower!");
    }
}
```

```
class Fungus extends Plant {
    no usages
    public Fungus(String name) {
        super(name);
    }
    no usages
    public void talk() {
        System.out.println("Hi! I'm " + name + "! And I'm a Fungus!");
    }
}
```

Wait, there is an error. Where do you think the error is?

Tips: You can see the error description by pressing 

on the top right corner. Then you will see where the error is located: 

Ah right! We have changed the Attribute “name” in the Plant class yesterday to private. Please make it public first!

What is the keyword “super” in the constructor above? You can learn it here.

After that, we can add 2 new objects in the main class. I made it like the picture below. (Please make it according to your creativity.)

```
public class Main {
    public static void main(String[] args) {
        Plant plant1 = new Flower( name: "Sunflower");
        Plant plant2 = new Fungus( name: "Sun-shroom");
        Plant plant3 = new Flower( name: "Marigold"); //new instance object
        Plant plant4 = new Fungus( name: "Puff-shoom"); //new instance object
        Farmer farmer1 = new Farmer( name: "Crazy Dave", plant1.getName());
        Farmer farmer2 = new Farmer( name: "Sober Dave", plant2.getName());
```

In the image above, I added 2 new objects:

1. An object named plant3 from the class plant with a child Flower.
2. An object named plant4 from the class plant with a child Fungus.
3. I changed the name in the constructor of plant2 to “Sun-shroom”, which was originally “Mushroom”

Then at the bottom of the code, I added the following code to call the talk() function on the Plant class with the Flower and Fungus children.

```
((Flower) plant1).talk();
((Fungus) plant2).talk();
((Flower) plant3).talk();
((Fungus) plant4).talk();
```

The line of code ((Flower) plant1).bloom(); is an example of downcasting in Java, which is converting a reference from the parent class (Plant) to its specific subclass (Flower). Once done, please run it:

```
Hello, World!
Current date and time: Mon Mar 10 14:22:27 WIB 2025
Hi! My name is: Crazy Dave. My favourite plant is: Sunflower
Hi! My name is: Sober Dave. My favourite plant is: Sun-shroom
Hi! I'm Sunflower! And I'm a Flower!
Hi! I'm Sun-shroom! And I'm a Fungus!
Hi! I'm Marigold! And I'm a Flower!
Hi! I'm Puff-shoom! And I'm a Fungus!

Process finished with exit code 0
```

Are you dizzy? If you are dizzy, please explore it yourself until you understand. Use the internet as an alternative source for your learning.

TIPS

Java inheritance:

Video

Materials

OVERRIDING

THEORY

Method overriding is a capability that allows you to replicate the body of the main method in the parent class. What makes the parent method different from the overriding method is the contents of the method.

In simple terms, method overriding is done when we want to recreate a method in a child class.

MATERIALS

Method overriding can be created by adding the @Override annotation above the method name or before creating the method. Example:

Parent class

```
public class parentClass {
    public void namaMethod() {
        System.out.println("From the parent class");
    }
}
```

Child class

```
class ChildClass extends ParentClass {
    @Override
    public void nameMethod() {
        System.out.println("This is an overridden method");
    }
}
```

An example of the use case of inheritance is, in the Hero class above there is a protect() method and we want to change the content of the method in the Superman child class to "protect the people of Earth from monster attacks". The protect() method in the Hero class does not need to be changed at all and remains like this:

```
public class Hero {
    private String name;

    public void protect() {
        System.out.println(name + " is protecting the community");
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

But in the Superman class we change it to something like this:

```java
public class Superman extends Hero {

    public void fly() {
        System.out.println(getName() + " is flying...");
    }

    @Override
    public void protect() {
        System.out.println(getName() + " protects the people of Earth from monster attacks");
    }
}
```

To find out the output of the changes, we can call it in the Main class in the same way as calling a regular method.

```java
public class Main {
    public static void main(String[] args) {
        // Object creation
        Spiderman spiderman = new Spiderman();
        Superman superman = new Superman();
        Deadpool deadpool = new Deadpool();

        // Calling methods from the parent class via child classes
        spiderman.setName("Tobey Maguire");
        superman.setName("Christopher");
        deadpool.setName("Ryan Reynolds");

        // Calling methods defined in the child classes
        spiderman.protect();
        superman.protect();
        deadpool.protect();
    }
}
```

It can be seen at the end of our code that we are calling the same method on 3 different objects, but the output will appear like this:

```
Tobey Maguire is protecting the community
Christopher is protecting the people of Earth from monster attacks
Ryan Reynolds is protecting the community

Process finished with exit code 0
```

This can happen because the Superman class has overridden the protect() method, so the contents of the method are no longer the same as the contents of the method in the parent class. Unlike the Spiderman and Deadpool classes that do not override, the contents will be the same as the protect() method in the parent class.

PRACTICE

Let's apply this Overriding concept in our project! Before we start, please answer this question: In which part of our project can we apply this overriding technique?

Yes! We can apply it to the talk() method in the Flower and Fungus classes, which are child classes of the parent class Plant. Please add a new method with the keyword @Override to the Plant class with the following specifications:

```
public void talk() {
    System.out.println("I am a plant named " + name);
}
```

Then we add the @Override annotation above the talk() method in the Flower and Fungus classes as follows:

```
@Override
public void talk() {
    System.out.println("Hi! I'm " + name + "! And I'm a Fungus!");
}
```

```
@Override
public void talk() {
    System.out.println("Hi! I'm " + name + "! And I'm a Flower!");
}
```

If so, then we have successfully overridden the talk() method. After that, we no longer need to downcast the talk() method call as in the previous practice. We can just write it like this:

before

```
((Flower) plant1).talk();
((Fungus) plant2).talk();
((Flower) plant3).talk();
((Fungus) plant4).talk();
```

after

```
plant1.talk();
plant2.talk();
plant3.talk();
plant4.talk();
```

If run the result will be the same as the previous practice. If you add a new object without using a child class. Then it will execute the talk() method in the Plant class. Example:

```java
public class Main {
    public static void main(String[] args) {
        Plant plant1 = new Flower( name: "Sunflower");
        Plant plant2 = new Fungus( name: "Sun-shroom");
        Plant plant3 = new Flower( name: "Marigold");
        Plant plant4 = new Fungus( name: "Puff-shoom");
        Plant plant5 = new Plant ( name: "Cactus"); //new instance object without child class
        Farmer farmer1 = new Farmer( name: "Crazy Dave", plant1.getName());
        Farmer farmer2 = new Farmer( name: "Sober Dave", plant2.getName());
```

Output:

```
Hello, World!
Current date and time: Mon Mar 10 15:31:05 WIB 2025
Hi! My name is: Crazy Dave. My favourite plant is: Sunflower
Hi! My name is: Sober Dave. My favourite plant is: Sun-shroom
Hi! I'm Sunflower! And I'm a Flower!
Hi! I'm Sun-shroom! And I'm a Fungus!
Hi! I'm Marigold! And I'm a Flower!
Hi! I'm Puff-shoom! And I'm a Fungus!
I am a plant named Cactus
```

TIPS

Short video explaining the overriding technique:

Video

Materials

SUPER KEYWORDS

THEORY

The super keyword is a keyword used to access attributes or methods of the parent class from the child class. In Java, this keyword can be used to do several things, such as the following:

- Accessing parent class methods or constructors by child classes
- Designates members of the parent class by the child class
- Accessing parent class methods by child classes

MATERIALS

For an example of using the super keyword, we try to make the Hero class have a constructor that is used to initialize the hero's name and age. Here is the code for each class:

File: Hero.java

```java
public class Hero {
    private String name;
    public int age;

    public Hero(String name, int age){
        this.name = name;
        this.age = age;
    }

    public void protect(){
        System.out.println(name + " protects the community");
    }

    public void setName(String name){
        this.name = name;
    }

    public String getName(){
        return name;
    }
}
```

We have changed the parent class with the constructor form Hero(String name, int age) so that the child class will be like this:

File: Spiderman.java

```java
public class Spiderman extends Hero {

    public Spiderman(String name, int age) {
        super(name, age);
    }

    public void climb() {
        System.out.println(getName() + " is climbing cliffs");
    }
}
```

File: Deadpool.java

```java
public class Deadpool extends Hero {

    public Deadpool(String name, int age) {
        super(name, age);
    }

    public void heal() {
        System.out.println(getName() + " is healing themselves");
    }
}
```

File: Superman.java

```java
public class Superman extends Hero {

    public Superman(String name, int age) {
        super(name, age);
    }

    public void fly() {
        System.out.println(getName() + " is flying...");
    }

    @Override
    public void protect() {
        System.out.println(getName() + " protects the people of Earth from monster attacks");
    }
}
```

It can be seen that the child classes Superman, Spiderman, and Deadpool are also required to have a constructor because the parent class has a constructor. But in the integrated child class constructor super(name, age); which is called, the purpose of super() is to call a method in the parent class where the method is the constructor of the Hero class. Can super() only be used on methods? The answer is no, it can also be used to call variables in the parent class through the child class. For example, in the Deadpool.java class we change it to be like this:

```java
public class Deadpool extends Hero {
    public Deadpool(String name, int age) {
        super(name, age);
    }

    public void heal() {
        System.out.println(super.getName() + " is healing themselves");
    }

    public void displayAge() {
        System.out.println(getName() + " is aged: " + super.age);
    }
}
```

In the Deadpool class above, we use the super keyword to call the age attribute and also the getName() method, where the age attribute and getName() method are not in the Deadpool class but are in the Hero class which is the parent class. It should be noted that for the use of the super keyword in the constructor, the call to the super() keyword must be at the beginning after the creation of the child class constructor to avoid errors.

The method of calling the Main class is as follows:

```java
public class Main {
    public static void main(String[] args) {
        // pembuatan object disertai dengan argumen yang sesuai
        Spiderman spiderman = new Spiderman("Tobey maguaire", 34);
        Superman superman = new Superman("Cristopher", 43);
        Deadpool deadpool = new Deadpool("Ryan Reynolds", 36);

        // coba panggil
        System.out.println(superman.getName());
        System.out.println(spiderman.getName());
        System.out.println(deadpool.getName());
    }
}
```

The output will be like this:

```
Cristopher
Tobey maguaire
Ryan Reynolds
```

TIPS

Material about super keywords:

Materials

Video

THIS KEYWORDS

THEORY

So far this material we have learned a lot about constructors, encapsulation, inheritance, overriding, super. Maybe in the middle or at the beginning of the material a question arises what is meant by the keyword this? The answer is that the keyword this is usually used to fill class variables or access variables in the class.

MATERIALS

Let's look at an example below:

```java
public class Hero {
    private String name;
    public int age;

    public Hero(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void protect() {
        System.out.println(name + " is protecting the community");
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }
}
```

In the Hero constructor and the setName() method, there is code that uses the keyword this. If you look closely, the Hero class has String name and int age attributes, in the Hero(String name, int age) constructor there are also attributes but as parameters. These two things are different for the private String name owned by the class while the String name in the constructor parameter is a parameter owned by the constructor. In the constructor this.name = name has a meaning that the name attribute which was initially empty is filled by the name parameter filled by the user.

This also applies to the setName() method, because this method is a setter or method used to initialize a private value, so this.name = name also means that the name attribute in the class is now filled by the name parameter, which is filled in when the setter method is called.

TIPS

Material about “this” keywords:

Materials

Video

CODELAB & TASK

CODELAB

Create a Java program that simulates a battle between a Hero and an Enemy using the concepts in Module 2.

The program must have the following classes:

1. GameCharacter Class (Superclass)
    - Has private attributes name and health.
    - Has getters and setters for the name and health attributes.
    - Has a method attack(GameCharacter target) that will be overridden by subclasses.
    - The constructor must accept name and health as parameters.
2. Hero Class (Subclass of GameCharacter)
    - Inherits GameCharacter and uses super() in the constructor.
    - Overrides the method attack(GameCharacter target), with the following effects:
        - Displays the message: "<Hero name> attacks <target name> using a sword!"
        - Reduces 20 health points from the target.
        - Displays the target's latest health.
3. Enemy Class (Subclass of GameCharacter)
    - Inherits GameCharacter and uses super() in the constructor.
    - Override the attack method(GameCharacter target), with the following effects:
        - Displays the message: "<Enemy name> attacked <target name> using magic!"
        - Reduces 15 health points from the target.
        - Displays the target's current health.
4. Main Class
    - Creates three objects:
        - A GameCharacter named "General Character" with 100 health.
        - A Hero named "Brimstone" with 150 health.

    - An Enemy named "Viper" with 200 health.
- Displays the initial health of the Hero and Enemy.
- Calls the attack() method to simulate combat:
    - Brimstone attacks Viper using Orbital Strike.
    - Viper attacks Brimstone using Snake Bite.

5. Example of expected output:

```
Initial status:
Brimstone has health: 150
Viper has health: 200
Brimstone attacks Viper using Orbital Strike!
Viper now has 180 health.
Brimstone attacks Viper using Orbital Strike!
Viper now has 160 health.
Viper attacks Brimstone using Snake Bite!
Brimstone now has 135 health.
```

TASK

Continue the simple login program in the previous module using the CONSTRUCTOR, ENCAPSULATION, INHERITANCE, OVERRIDING, SUPER KEYWORDS concepts in Java. This program must have three main classes, namely:

1. **User Class (as a superclass)**
    - Has encapsulated name and student ID attributes (private).
    - Has a constructor to initialize name and student ID.
    - Provides getters and setters for name and student ID attributes.
    - Has a login() method that will be overridden by subclasses.
    - Has a displayInfo() method to display user information.
2. **Admin Class (as a subclass of User)**
    - Has additional username and password attributes.

- Admin constructor uses super to initialize name and student ID.
- Overrides login() method to check username and password match.
- Overrides displayInfo() method to display successful login message.

3. **Student Class (as a subclass of User)**
    - Student constructor uses super to initialize name and student ID.
    - Override the login() method to match the input name and student ID.
    - Override the displayInfo() method to display a successful login message.
4. **LoginSystem class (as the main program)**
    - Provides login options as Admin or Student.
    - If the user selects Admin, the program asks for a username and password to login.
    - If the user selects Student, the program asks for a name and student ID to login.
    - If the login is successful, the user information will be displayed, if it fails, an error message will appear.

Implementation Instructions:

1. Use inheritance to create Admin and Student as subclasses of User.
2. Use overriding on the login() and displayInfo() methods.
3. Use encapsulation by applying the private modifier to attributes and providing getters & setters

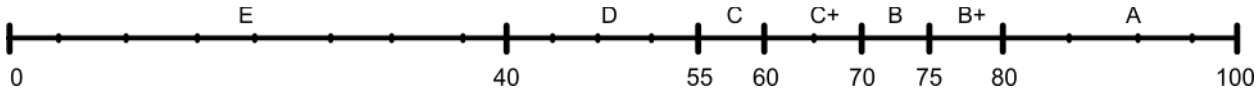
EVALUATION

ASSESSMENT RUBRIC

Assessment Aspects	Point
CODELAB	**Total 30%**
Code neatness	5%
Code & output accuracy	15%
Code creativity	5%
Original code (not copied)	5%

TASK	Total 70%
Code neatness	5%
Code & output accuracy	20%
Original code (not copied)	5%
Ability to explain	20%
Answering questions	20%
TOTAL	**100%**

ASSESSMENT SCALE



A = (81 - 100) → Excellent

B+ = (75 - 80) → Very Good

B = (70 - 74) → Good

C+ = (60 - 69) → Fairly Good

C = (55 - 59) → Fair

D = (41 - 54) → Poor

E = (0 - 40) → Bro really...

END MODULE SUMMARY

In this module, we have touched a little bit on Polymorphism. What is polymorphism? You will learn more about it in the next module. In essence, polymorphism allows you to program in a more abstract yet efficient way.

Understanding or paradigms in programming will be easier to understand if we directly practice and experiment rather than just reading the material. Therefore, abstract thinking skills and good imagination are very necessary to master concepts like this.

Don't forget, learning is a journey. It's normal to make mistakes, in fact, from your mistakes you will understand more. For example, if your score in the previous module was BAD, you can still maximize it in the next modules, or in the theory class. So, stay enthusiastic and keep exploring. Who knows, from what you learn now, it can make you a reliable programmer.