Multiple Class, Attribute, Object

ILMU KOMPUTER UNIV. PERTAMINA

Create a Class

To create a class, use the keyword `class`:

MyClass.java

Create a class named "`MyClass`" with a variable x:

```
public class MyClass {
  int x = 5;
}
```

Create an Object

In Java, an object is created from a class. We have already created the class named `MyClass`, so now we can use this to create objects.

To create an object of `MyClass`, specify the class name, followed by the object name, and use the keyword `new`:

Example

Create an object called "`myObj`" and print the value of x:

```
public class MyClass {
  int x = 5;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    System.out.println(myObj.x);
  }
}
```

Multiple Objects

You can create multiple objects of one class:

Example

Create two objects of `MyClass` :

```
public class MyClass {
  int x = 5;

  public static void main(String[] args) {
    MyClass myObj1 = new MyClass();  // Object 1
    MyClass myObj2 = new MyClass();  // Object 2
    System.out.println(myObj1.x);
    System.out.println(myObj2.x);
  }
}
```

Using Multiple Classes

You can also create an object of a class and access it in another class. This is often used for better organization of classes (one class has all the attributes and methods, while the other class holds the `main()` method (code to be executed)).

Remember that the name of the java file should match the class name. In this example, we have created two files in the same directory/folder:

- MyClass.java
- OtherClass.java

MyClass.java

```
public class MyClass {
  int x = 5;
}
```

MyClass.java

```java
public class MyClass {
  int x = 5;
}
```

OtherClass.java

```java
class OtherClass {
  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    System.out.println(myObj.x);
  }
}
```

- Compile both file
- Run OtherClass.java
- Output: 5

Accessing Attributes

You can access attributes by creating an object of the class, and by using the dot syntax (`.`):

The following example will create an object of the `MyClass` class, with the name `myObj` . We use the `x` attribute on the object to print its value:

Example

Create an object called " `myObj` " and print the value of `x` :

```
public class MyClass {
  int x = 5;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    System.out.println(myObj.x);
  }
}
```

Modify Attributes

You can also modify attribute values:

Example

Set the value of `x` to 40:

```
public class MyClass {
  int x;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    myObj.x = 40;
    System.out.println(myObj.x);
  }
}
```

Or override existing values:

Example

Change the value of `x` to 25:

```
public class MyClass {
  int x = 10;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    myObj.x = 25; // x is now 25
    System.out.println(myObj.x);
  }
}
```

If you don't want the ability to override existing values, declare the attribute as `final`:

Example

```
public class MyClass {
  final int x = 10;

  public static void main(String[] args) {
    MyClass myObj = new MyClass();
    myObj.x = 25; // will generate an error: cannot assign a value to a final
    System.out.println(myObj.x);
  }
}
```

The final keyword is useful when you want a variable to always store the same value, like PI (3.14159...).

The final keyword is called a "modifier"

Multiple Objects

If you create multiple objects of one class, you can change the attribute values in one object, without affecting the attribute values in the other:

Example

Change the value of `x` to 25 in `myObj2`, and leave `x` in `myObj1` unchanged:

```
public class MyClass {
  int x = 5;

  public static void main(String[] args) {
    MyClass myObj1 = new MyClass();  // Object 1
    MyClass myObj2 = new MyClass();  // Object 2
    myObj2.x = 25;
    System.out.println(myObj1.x);  // Outputs 5
    System.out.println(myObj2.x);  // Outputs 25
  }
}
```

Multiple Attributes

You can specify as many attributes as you want:

Example

```
public class Person {
  String fname = "John";
  String lname = "Doe";
  int age = 24;

  public static void main(String[] args) {
    Person myObj = new Person();
    System.out.println("Name: " + myObj.fname + " " + myObj.lname);
    System.out.println("Age: " + myObj.age);
  }
}
```

Java Class Methods

methods are declared within a class, and that they are used to perform certain actions:

Example

Create a method named `myMethod()` in MyClass:

```
public class MyClass {
  static void myMethod() {
    System.out.println("Hello World!");
  }
}
```

`myMethod()` prints a text (the action), when it is **called**. To call a method, write the method's name followed by two parentheses **()** and a semicolon**;**

Variable Scope

- Variable access
  - Lokal : in method, hanya bisa diakses di method yang mendeklarasikan variable tersebut
  - Instance: in class, access with object in main
  - Static: in class, access anywhere
- Reference variable and object
- Primitive and non-primitive object

Local

```
public class StudentDetails {
    public void StudentAge()
    {
        // local variable age
        int age = 0;
        age = age + 5;
        System.out.println("Student age is : " + age);
    }

    public static void main(String args[])
    {
        StudentDetails obj = new StudentDetails();
        obj.StudentAge();
    }
}
```

Local Variables: A variable defined within a block or method or constructor is called local variable

Tidak bisa diakses dari main

Output:
Student age is : 5

Local

```
public class StudentDetails {
    public void StudentAge()
    { // local variable age
        int age = 0;
        age = age + 5;
    }

    public static void main(String args[])
    {
        // using local variable age outside it's scope
        System.out.println("Student age is : " + age);
    }
}
```

```
Compilation Error in java code :-
prog.java:12: error: cannot find symbol
        System.out.println("Student age is : " + age);
                                                ^
  symbol:   variable age
  location: class StudentDetails
1 error
```

```java
class Marks {
    // These variables are instance variables.
    // These variables are in a class
    // and are not inside any function
    int engMarks;
    int mathsMarks;
    int phyMarks;
}

class MarksDemo {
    public static void main(String args[])
    {
        // first object
        Marks obj1 = new Marks();
        obj1.engMarks = 50;
        obj1.mathsMarks = 80;
        obj1.phyMarks = 90;

        // second object
        Marks obj2 = new Marks();
        obj2.engMarks = 80;
        obj2.mathsMarks = 60;
        obj2.phyMarks = 85;
```

Instance Variables

- **Instance Variables**: Instance variables are non-static variables and are declared in a class outside any method, constructor or block.
- Bisa diakses dari main melalui objek

```java
        // displaying marks for first object
        System.out.println("Marks for first object:");
        System.out.println(obj1.engMarks);
        System.out.println(obj1.mathsMarks);
        System.out.println(obj1.phyMarks);

        // displaying marks for second object
        System.out.println("Marks for second object:");
        System.out.println(obj2.engMarks);
        System.out.println(obj2.mathsMarks);
        System.out.println(obj2.phyMarks);
    }
}
```

static variables

```
class Emp {

    // static variable salary
    public static double salary;
    public static String name = "Harsh";
}

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

        // accessing static variable without object
        Emp.salary = 1000;
        System.out.println(Emp.name + "'s average salary:"
                           + Emp.salary);
    }
}
```

To access static variables in main,

No need to create an object of that class,

can simply access the variable as:

class_name.variable_name;

```
Harsh's average salary:1000.0
```

`myMethod()` prints a text (the action), when it is **called**. To call a method, write the method's name followed by two parentheses **()** and a semicolon**;**

Example

Inside `main`, call `myMethod()`:

```
public class MyClass {
  static void myMethod() {
    System.out.println("Hello World!");
  }

  public static void main(String[] args) {
    myMethod();
  }
}

// Outputs "Hello World!"
```

Static vs. Non-Static

You will often see Java programs that have either `static` or `public` attributes and methods.

In the example above, we created a `static` method, which means that it can be accessed without creating an object of the class, unlike `public`, which can only be accessed by objects:

An example to demonstrate the differences between `static` and `public` **methods**:

```
public class MyClass {
  // Static method
  static void myStaticMethod() {
    System.out.println("Static methods can be called without creating objects
  }

  // Public method
  public void myPublicMethod() {
    System.out.println("Public methods must be called by creating objects");
  }

  // Main method
  public static void main(String[] args) {
    myStaticMethod(); // Call the static method
    // myPublicMethod(); This would compile an error

    MyClass myObj = new MyClass(); // Create an object of MyClass
    myObj.myPublicMethod(); // Call the public method on the object
  }
}
```

Create a Car object named `myCar`. Call the `fullThrottle()` and `speed()` methods on the `myCar` object, and run the program:

```java
// Create a Car class
public class Car {

  // Create a fullThrottle() method
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  // Create a speed() method and add a parameter
  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }

  // Inside main, call the methods on the myCar object
  public static void main(String[] args) {
    Car myCar = new Car();      // Create a myCar object
    myCar.fullThrottle();       // Call the fullThrottle() method
    myCar.speed(200);           // Call the speed() method
  }
```

The dot (`.`) is used to access the object's attributes and methods.

To call a method in Java, write the method name followed by a set of parentheses **()**, followed by a semicolon (`;`).

A class must have a matching filename (`Car` and **Car.java**).

Access Method with An Object

Output:
The car is going as fast as it can!
Max speed is: 200

Using Multiple Classes

example, we have created two files in the same directory:

- Car.java
- OtherClass.java

Car.java

```
public class Car {
  public void fullThrottle() {
    System.out.println("The car is going as fast as it can!");
  }

  public void speed(int maxSpeed) {
    System.out.println("Max speed is: " + maxSpeed);
  }
}
```

OtherClass.java

```
class OtherClass {
  public static void main(String[] args) {
    Car myCar = new Car();      // Create a myCar object
    myCar.fullThrottle();       // Call the fullThrottle() method
    myCar.speed(200);           // Call the speed() method
  }
}
```

-Compile Both
-run main program

Output:
The car is going as fast as it can!
Max speed is: 200

Java Constructors

A constructor in Java is a **special method** that is used to initialize objects. The constructor is called when an object of a class is created. It can be used to set initial values for object attributes:

Example

Create a constructor:

```
// Create a MyClass class
public class MyClass {
  int x;  // Create a class attribute

  // Create a class constructor for the MyClass class
  public MyClass() {
    x = 5;  // Set the initial value for the class attribute x
  }

  public static void main(String[] args) {
    MyClass myObj = new MyClass(); // Create an object of class MyClass
    System.out.println(myObj.x); // Print the value of x
  }
}
```

Note that the constructor name must **match the class name**, and it cannot have a **return type** (like `void`).

Also note that the constructor is called when the object is created.

All classes have constructors by default: if you do not create a class constructor yourself, Java creates one for you. However, then you are not able to set initial values for object attributes.

Constructor Parameters

Constructors can also take parameters, which is used to initialize attributes.

The following example adds an `int y` parameter to the constructor. Inside the constructor we set x to y (x=y). When we call the constructor, we pass a parameter to the constructor (5), which will set the value of x to 5:

```
public class MyClass {
  int x;

  public MyClass(int y) {
    x = y;
  }

  public static void main(String[] args) {
    MyClass myObj = new MyClass(5);
    System.out.println(myObj.x);
  }
}
```

Add Many Parameter

```
public class Car {
  int modelYear;
  String modelName;

  public Car(int year, String name) {
    modelYear = year;
    modelName = name;
  }

  public static void main(String[] args) {
    Car myCar = new Car(1969, "Mustang");
    System.out.println(myCar.modelYear + " " + myCar.modelName);
  }
}

// Outputs 1969 Mustang
```

Modifiers

By now, you are quite familiar with the `public` keyword that appears in almost all of our examples:

```
public class MyClass
```

The `public` keyword is an **access modifier**, meaning that it is used to set the access level for classes, attributes, methods and constructors.

We divide modifiers into two groups:

- **Access Modifiers** - controls the access level
- **Non-Access Modifiers** - do not control access level, but provides other functionality

Access Modifier

For **attributes, methods and constructors**, you can use the one of the following:

Modifier	Description
`public`	The code is accessible for all classes
`private`	The code is only accessible within the declared class
default	The code is only accessible in the same package. This is used when you don't specify a modifier.
`protected`	The code is accessible in the same package and **subclasses**.

Package

- A package in Java is used to group related classes. Think of it as **a folder in a file directory**.
- We use packages to avoid name conflicts, and to write a better maintainable code. Packages are divided into two categories:
  - Built-in Packages (packages from the Java API)
  - User-defined Packages (create your own packages)

Package

The library is divided into **packages** and **classes**. Meaning you can either import a single class (along with its methods and attributes), or a whole package that contain all the classes that belong to the specified package.

Syntax

```
import package.name.Class;   // Import a single class
import package.name.*;   // Import the whole package
import java.util.Scanner;
import java.util.*;
```

Here:
→ **java** is a top level package
→ **util** is a sub package
→ and **Scanner** is a class which is present in the sub package **util**.

To use the `Scanner` class, create an object of the class and use any of the available methods found in the `Scanner` class documentation. In our example, we will use the `nextLine()` method, which is used to read Strings:

Example

```
import java.util.Scanner;  // Import the Scanner class

class MyClass {
  public static void main(String[] args) {
    Scanner myObj = new Scanner(System.in);  // Create a Scanner object
    System.out.println("Enter username");

    String userName = myObj.nextLine();  // Read user input
    System.out.println("Username is: " + userName);  // Output user input
  }
}
```

Input Types

In the example above, we used the `nextLine()` method, which is used to read Strings. To read other types, look at the table below:

Method	Description
`nextBoolean()`	Reads a `boolean` value from the user
`nextByte()`	Reads a `byte` value from the user
`nextDouble()`	Reads a `double` value from the user
`nextFloat()`	Reads a `float` value from the user
`nextInt()`	Reads a `int` value from the user
`nextLine()`	Reads a `String` value from the user
`nextLong()`	Reads a `long` value from the user
`nextShort()`	Reads a `short` value from the user

User-defined Packages

- To create your own package, you need to understand that Java uses a file system directory to store them. Just like folders on your computer
  - └── root
  - └── mypack
  - └── MyPackageClass.java

To create a package, use the `package` keyword:

MyPackageClass.java

```
package mypack;
class MyPackageClass {
  public static void main(String[] args) {
    System.out.println("This is my package!");
  }
}
```

Contoh

- created a class Calculator inside a package name letmecalculate.
- To create a class inside a package, declare the package name in the first statement in your program.
- A class can have only one package declaration.

Calculator.java file created inside a package letmecalculate

```
package letmecalculate;

public class Calculator {
    public int add(int a, int b){
        return a+b;
    }
    public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(10, 20));
    }
}
```

Contoh

Now lets see how to use this package in another program.

```
import letmecalculate.Calculator;
public class Demo{
   public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(100, 200));
   }
}
```

To use the class Calculator, I have imported the package letmecalculate. In the above program I have imported the package as letmecalculate.Calculator, this only imports the Calculator class. However if you have several classes inside package letmecalculate then you can import the package like this, to use all the classes of this package.

import letmecalculate.*;

Creating a class inside package while importing another package

As we have seen that both package declaration and package import should be the first statement in your java program.

```java
//Declaring a package
package anotherpackage;
//importing a package
import letmecalculate.Calculator;
public class Example{
   public static void main(String args[]){
        Calculator obj = new Calculator();
        System.out.println(obj.add(100, 200));
   }
}
```

So the order in this case should be:

→ package declaration

→ package import

Level	Modifier	Same Class	Same Package	Subclass	Universe
Private	Private	YES			
Default		YES	YES		
Protected	protected	YES	YES	YES	
Public	public	YES	YES	YES	YES

Private access modifier

- The scope of private modifier is limited to the class only.
- Private Data members and methods are only accessible within the class
- Class and **Interface** cannot be declared as private
- If a class has **private constructor** then you cannot create the object of that class from outside of the class.

Private access modifier example in java

This example throws compilation error because we are trying to access the private data member and method of class ABC in the class Example. The private data member and method are only accessible within the class.

```
class ABC{
   private double num = 100;
   private int square(int a){
        return a*a;
   }
}
public class Example{
   public static void main(String args[]){
        ABC obj = new ABC();
        System.out.println(obj.num);
        System.out.println(obj.square(10));
   }
}
```

Output:

```
Compile - time error
```

Default

- When we do not mention any access modifier, it is called default access modifier.
- The scope of this modifier is limited to the package only. This means that if we have a class with the default access modifier in a package, only those classes that are in this package can access this class. No other class outside this package can access this class.
- Similarly, if we have a default method or data member in a class, it would not be visible in the class of another package.

Default

In this example we have two classes, Test class is trying to access the default method of Addition class, since class Test belongs to a different package, this program would throw compilation error, because the scope of default modifier is limited to the same package in which it is declared.

Addition.java

```
package abcpackage;


public class Addition {
   /* Since we didn't mention any access modifier here, it would
    * be considered as default.
    */
   int addTwoNumbers(int a, int b){
        return a+b;
   }
}
```

Default Cont.

Test.java

```java
package xyzpackage;

/* We are importing the abcpackage
 * but still we will get error because the
 * class we are trying to use has default access
 * modifier.
 */
import abcpackage.*;
public class Test {
   public static void main(String args[]){
        Addition obj = new Addition();
        /* It will throw error because we are trying to access
         * the default method in another package
         */
        obj.addTwoNumbers(10, 21);
   }
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
The method addTwoNumbers(int, int) from the type Addition is not visible
at xyzpackage.Test.main(Test.java:12)
```

Protected Access Modifier

Protected data member and method are only accessible by the classes of the same package and the subclasses present in any package. You can also say that the protected access modifier is similar to default access modifier with one exception that it has visibility in sub classes. Classes cannot be declared protected. This access modifier is generally used in a parent child relationship.

Protected access modifier example in Java

- In this example the class Test which is present in another package is able to call the addTwoNumbers() method, which is declared protected. This is because the Test class extends class Addition and the protected modifier allows the access of protected members in subclasses (in any packages).

Addition.java

```
package abcpackage;
public class Addition {

   protected int addTwoNumbers(int a, int b){
        return a+b;
   }
}
```

Protected access modifier example in Java

Test.java

```
package xyzpackage;
import abcpackage.*;
class Test extends Addition{
   public static void main(String args[]){
        Test obj = new Test();
        System.out.println(obj.addTwoNumbers(11, 22));
   }
}
```

Output:

```
33
```

Public access modifier

- The members, methods and classes that are declared public can be accessed from anywhere. This modifier doesn’t put any restriction on the access.

public access modifier example in java

Lets take the same example that we have seen above but this time the method addTwoNumbers() has public modifier and class Test is able to access this method without even extending the Addition class. This is because public modifier has visibility everywhere.

Addition.java

```
package abcpackage;

public class Addition {

   public int addTwoNumbers(int a, int b){
        return a+b;
   }
}
```

Test.java

```
package xyzpackage;
import abcpackage.*;
class Test{
   public static void main(String args[]){
      Addition obj = new Addition();
      System.out.println(obj.addTwoNumbers(100, 1));
   }
}
```

Output:

```
101
```

Access Modifiers

For **classes**, you can use either `public` or *default*:

Modifier	Description
`public`	The class is accessible by any other class
default	The class is only accessible by classes in the same package. This is used when you don't specify a modifier.

Non-Access Modifiers

For **classes**, you can use either `final` or `abstract`:

Modifier	Description
`final`	The class cannot be inherited by other classes
`abstract`	The class cannot be used to create objects (To access an abstract class, it must be inherited from another class.

Non-Access Modifier

For **attributes and methods**, you can use the one of the following:

Modifier	Description
`final`	Attributes and methods cannot be overridden/modified
`static`	Attributes and methods belongs to the class, rather than an object
`abstract`	Can only be used in an abstract class, and can only be used on methods. The method does not have a body, for example **abstract void run();**. The body is provided by the subclass (inherited from).