**Belajar java**

**Pengenalan java**

Dibuat oleh james gosling pada 1995. Java merupakan pemrograman berorientasi objek dan mendukung pengelolaan memori secara otomatis. Java dikenal dengan write once, run anywhere. Artinya Ketika sebuah code java dibuat dan decompile akan menghasilkan sebuah binary program. Nah binary program ini itu independent tidak terkait dengan system operasi. Untuk menjalankan binary java menggunakan jvm (java virtual machine).

JRE (java runtime environment) (Ketika akan melakukan deploying)

JDK (java development kit) (masih tahap pengembangan)

Proses development program java

Ketika kita menuliskan code java dengan extension File.java Ketika decompile akan menghasilklan java binary (File.class) file ini lah yang nanti akan diproses oleh JRE. Namun Ketika File.class sudah banyak maka akan disatukan menggunakan java archive (File.jar) yang nantinya JRE akan memprosesnya.

**Instalasi java**

Download jdk di openjdk karena free dan opensource. Bisa didownload di dalam jdk.java.net. dan dalam environment variable ubah JAVA\_HOME ke path folder yang sudah di download. Dan path nya ke bin di dalam folder java tersebut.

Ketika memasuki pertama kali intellij idea terdapat proses indexing yaitu untuk memudahkan IDE dalam pencarian file.

Tipe data di java

Variable dalam java yaitu static type yaitu Ketika membuat variable harus ditentukan tipe datanya jadi tidak dinamis seperti javascript.

Tipe data terbagi menjadi dua primitive dan non primitive. Primitive yaitu tipe data bawaan di program seperti int, char, Boolean sedangkan non primitive seperti String, Integer, Long, Float, Double. Note tipe data primitive selalu memiliki default value sedangkan non primitive default value selalu null. Non primitive data type mempunyai method atau function untuk memanipulasi variable tersebut.

Integer number

Byte (1 byte) -128 -> 127  
Short (2 byte) -32,769 -> 32, 767  
Int (4 byte) 2.10 pangkat 9  
Long (8 byte) 2.10 pangkat 18

Float (4 bytes) 3.4 pangkat 038  
Double (8 bytes) 1.7 pangkat 308

// kode literal  
int decimalInt = 10;  
int hexInt = 0xFFFFFF;  
int binaryDecimal = 0b10101010;  
  
// for readibiilty  
int satuJuta = 1\_000\_000;

Char merupakan tipe data untuk satu karakter saja   
Bool binary tipe data yang berisi true dan false (default value = false)

char firstWord = 'a'; // harus dengan single tik  
  
System.*out*.println(firstWord);  
  
boolean iniBool = false;

Tipedata non primitive

Integer angka = 5;  
Long angkaLong = 100L;  
  
Byte iniByte;  
iniByte = 50;

Array

// kumpulan tipe data yang sama  
// jumlah array tidak bisa dirubah ketika pertama kali diinisialisasi  
  
// 1st method to declare an array  
String[] stringArray;  
stringArray = new String[3];  
  
stringArray[0] = "Mujahid ";  
stringArray[1] = "Ansori ";  
stringArray[2] = "Majid ";  
  
System.*out*.println(stringArray[1]);  
  
int[] intArray = new int[]{  
 10,10,10  
};  
  
long[] longArray = {10L, 10L};  
  
System.*out*.println(intArray.length);  
  
// 2 dimensions array  
  
String[][] members = {  
 {"Mujahid", "Ansori"},  
 {"Farhan", "Nurzaman"},  
 {"Hilmy", "Apriadi"}  
  
};  
  
System.*out*.println(members[0][1]); // ansori

Expression = a piece of code that result a value;  
statement = instruction or action (condition and loops);

We can’t put a statement when an expression is expected

Example  
System.out.println(const x); // error  
var x = if(a < 2) { return 2}; // error

Percabganan

var absen = 70;  
var ujianAkhir = 75;  
  
if(absen >= 75 && ujianAkhir >= 75){  
 System.*out*.println("Selamat Anda lulus");  
}  
  
char nilai = 'A';  
switch(nilai){  
 case 'A':  
 System.*out*.println("lulus dengan sangat baik");  
 break;  
 case 'B':  
 System.*out*.println("lulus dengan sangat baik");  
 break;  
 case 'C':  
 System.*out*.println("lulus dengan sangat baik");  
 break;  
 case 'D':  
 System.*out*.println("lulus dengan sangat baik");  
 break;  
 case 'E':  
 System.*out*.println("lulus dengan sangat baik");  
 break;  
 default:  
 System.*out*.println("jelek banget ku");  
  
}

// dimulai pada java 14 kita dapat menggunakan lambda dalam swith case  
switch (nilai){  
 case 'A' ->System.*out*.println("bagus banget gila");  
 case 'B', 'C' -> System.*out*.println("gila sih");  
 default -> {System.*out*.println("jelek");}  
}  
  
// menggunakan yeild  
String ucapan = switch (nilai) {  
 case 'A':  
 yield "Bagus banget gila";  
 case 'B', 'C':  
 yield "mayanlah";  
 default:  
 yield "jelek banget gila";  
};  
  
System.*out*.println(ucapan);

Perulangan

for(int i=0;i<5;i++){  
 for(int j=0;j<i;j++){  
 System.*out*.print("\*");  
 }  
 System.*out*.println();  
}  
  
var nilai = 50;  
boolean masihRemed = nilai >= 70 ? false:true;  
while(masihRemed){  
 System.*out*.println("remedke");  
 nilai+=5;  
 masihRemed = nilai >= 70 ? false:true;  
}  
  
do{  
 System.*out*.println("akan dijalankan meskipun kondisi tidka terpenuhi");  
} while(masihRemed);  
  
String[] nama = {  
 "Mujahid", "ansori", "majid"  
};  
  
for (var name: nama  
) {  
 System.*out*.println(name);  
}

method

public static void main(String[] args) {  
 *sayHelloWorld*();  
 System.*out*.println(*addTwoNumbers*(2,3));  
  
 // tanpa variable arguments  
 int[] values = {80,80};  
 *sayCongrats*("Mujahid", values);  
 *sayCongratsVarargs*("Mujahid", 80, 80);  
}  
static void sayHelloWorld(){  
 System.*out*.println("hello world");  
}  
static int addTwoNumbers(int a, int b){  
 return a + b;  
}  
// method variable arguments, jumlah argument yang tidak tentu  
// harus berada di akhir  
static void sayCongrats(String name, int[] values){  
 int total = 0;  
 for(var value: values){  
 total += value;  
 }  
  
 int finalValue = total / values.length;  
 if(finalValue >= 75){  
 System.*out*.println("Selamat " + name + " Anda lulus");  
 } else {  
 System.*out*.println("Mohon maaf anda tidak lulus");  
 }  
}  
  
static void sayCongratsVarargs(String name, int... values){  
 int total = 0;  
 for(var value: values){  
 total += value;  
 }  
  
 int finalValue = total / values.length;  
 if(finalValue >= 75){  
 System.*out*.println("Selamat " + name + " Anda lulus");  
 } else {  
 System.*out*.println("Mohon maaf anda tidak lulus");  
 }  
}

**Overloading =** method dengan nama yang sama dengan parameter yang berbeda.

sebuah block program yang akan dijalankan Ketika dipanggil

**OOP dalam java**

Class = prototype, blueprint  
object = ya object yang dibuat dari class yang telah dibuat.

Di dalam sebuah class terdapat dua macam yang esensial, field dan method. Field ya kaya variable tapi nanti Ketika oboject dibuat harus dicantumkan

Variable shadowing = merupakan kejadian Ketika kita membuat nama variable dengan nama yang sama di scope yang menutupi variable dengan nama yang sama di scope di atasnya.

This = mengakses object yang saat ini digunakan atau si classnya. Penggunaan this bisa untuk ngerefer attribute dalam class jika nama field dalam class sama dengan parameter di dalam method. This juga bisa digunakan untuk memanggil constructor lain.

**Inheritance =** penurunan suatu class ke kelass lain (parent and child) sebuah child hanya memiliki satu parent (dalam java). Namun parent dapat memiliki banyak child class. Ketika parent class mempunyai child class maka semua behaviour parent class akan turun kepada child classnya.

Overloading = mendelkarasikan sebuah method dengan parameter yang berbeda  
overriding = mendekrasikan ulang sebuah method yang sudah ada di parent class. Yang membuat method yang ada di parent tidak bisa diakses lagi melalui cchild class (tapi bisa sih menggunakan super keyword).

Person(String name, String address){  
 // name = name; variable shadowing  
 // makanya menggunakan this untuk nge refer attribute si object bukan si parameter  
  
 this.name = name;  
 this.address = address;  
 }  
  
 Person(String name) {  
// this.name = name;  
// this.address = "the address has not been initialize";  
 // memanggil constructor lain  
 this(name, "the address has not been initialize");  
 }

Super keyword = digunakan untk mengakses parent method, field, constructor menggunakan class child yang telah mengoverride. Child class harus mengoveride constructor parentnya. Jika sama maka bisa menggunakan super keyword. Yang wajibnya hanya memanggil satu constructor (jika ada 3 method overloading pada parent) maka wajibny ahanya 1.

class Shape {  
 int getCorner() {  
 return 0;  
 }  
}  
  
class Rectangle extends Shape {  
 int getCorner() {  
 return 4;  
 }  
  
 // memanggil getCorner yang ada di shape (parentnya)  
 int getParentCorner() {  
 return super.getCorner();  
 }  
}

setiap class akan menjadi turunan dari class “Object” meskipun tidak secara explicit menuliskan extends Object. Contoh method dari class Object yaitu (getClass, hashCode, equals, clone, toString, notify dll).

Polymorpishm (banyak bentuk) = kemampuan sebuah object berubah bentuk dengan bentuk lain. Polimorpisme sangat erat kaitannya dengan inheritance. Polimorpisme dapat digunakan Ketika kita mempunya sebuah method dengan parameter parent class. Maka parameter tersebut dapat dimasuki oleh child classnya

Type check and casting.

Seperti yang sudah dipraktekan dalam tipe data primitive perihal casting. Non primitive juga dapat dilkakukan casting. Namun agar aman harus diperiksa dahulu.

**Access modifier**

**Public, protected, no modifier, private**

Public = class, package, subclass, world;  
protected = class, package, subclass;  
no modifier = class, package;  
private = class;

1 public class di 1 file java.  
public class harus sama dengan nama file.

**Import**

Import merupakan mengambil class dari package lain. Syarat class yang harus diimport itu harus public. Untuk mengimport semua class yang ada di sebuah package itu bisa menggunakan Bintang (\*). FYI untuk semua class di java.lang itu sudah otomatis diimport secara explisit.

**Abstract class**

Abstract class merupakan sebuah class yang tidak bisa dibuat object secara langsung tapi hanya bisa diturunkan kepada class lain. Untuk class yang menjadi child class dari sebuah abstract class, itu harus memenuhi semua attribute atau method yang ada di abstract class. Dalam pembuatan method di abstract class itu tidak boleh ada body method. Karena bertujuan untuk dioverride oleh child classnya. Dan juga untuk pembuatan method di abstract class tidak boleh private.

**Encapsulation**

Encapsulation artinya memastikan data sensitive sebuah object tersembunyi dari akses luar, bertujuan agar kitab isa menjaga agar data sebuah object tetap baik dan valid. Untuk mencapai tersebut kita bisa menggunakan access modifier private. Dan untuk mendapatkan data tersebut kita bisa menggunakan getter dan setter.

Encapsulation is a fundamental concept in object-oriented programming (OOP) that is employed in languages like Java to ensure data privacy and control access to the internal state of an object. It involves bundling data (attributes or fields) and methods (functions) that operate on that data into a single unit, called a class. The main reasons why encapsulation exists in Java are:

1. **Data Hiding and Privacy:** Encapsulation allows you to hide the internal details of an object's state from the outside world. This means that the data within an object can only be accessed and modified through well-defined methods (getters and setters) provided by the class. This prevents unauthorized access and manipulation of an object's data, ensuring its integrity and consistency.
2. **Controlled Access:** By using access modifiers like private, protected, and public, Java allows you to control which parts of a class are accessible from outside the class. Private members can only be accessed within the class itself, while protected members can be accessed within the class and its subclasses. Public members are accessible from anywhere.
3. **Flexibility and Maintenance:** Encapsulation promotes a clear separation between the internal implementation details of a class and its external interface. This separation allows you to modify the internal implementation without affecting the classes that use the object. This is crucial for maintaining code flexibility and making changes without causing widespread disruptions.
4. **Code Organization:** Encapsulation helps in organizing your code by grouping related data and behaviors together within a class. This makes the codebase more structured, easier to understand, and more maintainable, especially as the codebase grows larger.
5. **Code Reusability:** Encapsulation allows you to create reusable components (classes) that can be used in various parts of your application or in different projects. This promotes the principle of Don't Repeat Yourself (DRY) by encapsulating common functionalities into easily reusable modules.
6. **Validation and Control:** With encapsulation, you can enforce validation checks and business logic within the methods that modify the object's internal state. This ensures that data entering the object adheres to certain rules, maintaining data integrity and consistency.

In Java, encapsulation is achieved by using access modifiers to specify the visibility of fields and methods, and by providing getter and setter methods to control how external code interacts with the internal state of objects. By embracing encapsulation, you create more robust and maintainable code that is less prone to bugs and easier to collaborate on within a team environment.

is it for safety?

Yes, encapsulation in Java serves the purpose of enhancing safety and security in your code. Here's how encapsulation contributes to safety:

1. **Data Integrity:** Encapsulation ensures that the internal state of an object can only be modified through controlled methods (getters and setters). This prevents unintended or unauthorized changes to the object's data. By enforcing validation and business logic within these methods, you can maintain data integrity and prevent invalid or inconsistent data states.
2. **Access Control:** Access modifiers (such as private, protected, and public) enable you to control which parts of a class are accessible from outside the class. Private members are hidden from outside access, which prevents other classes from directly modifying sensitive data. This access control minimizes the risk of unintended interference with an object's state.
3. **Hidden Implementation Details:** Encapsulation hides the implementation details of a class. This prevents other parts of the program from relying on these implementation specifics, which can change over time without affecting the external interface. This isolation reduces the chances of unexpected side effects when the implementation changes.
4. **Preventing Inadvertent Errors:** When accessing an object's state only through well-defined methods, you reduce the chances of making inadvertent mistakes in data manipulation. This can prevent subtle bugs and errors that might arise from directly modifying data fields without adhering to the proper validation checks or business rules.
5. **Code Maintenance and Refactoring:** Encapsulation makes it easier to maintain and refactor your codebase. When the internal structure of a class is encapsulated, you can modify its implementation without affecting the rest of the codebase that uses the class. This isolation reduces the risk of introducing new bugs or breaking existing functionality.
6. **Collaborative Development:** Encapsulation enhances collaboration in team development environments. By hiding internal implementation details and exposing only a well-defined interface, team members can interact with classes and objects without needing to understand their intricate inner workings. This promotes modular development and makes it easier for team members to work on different parts of the codebase simultaneously.

In summary, while encapsulation primarily focuses on data hiding and controlling access, it also contributes significantly to the safety and robustness of your code by preventing unintended modifications, reducing the impact of changes, and enhancing overall code reliability.

can you give me an example of vulnarability the code without encapsulation

Certainly, here's an example that demonstrates a vulnerability in code without proper encapsulation:

Let's say you're working on a simple banking application where customers have a balance that can be updated using methods. Here's how a class might look without proper encapsulation:

public class BankAccount {

public String accountHolder;

public double balance;

public void deposit(double amount) {

balance += amount;

}

public void withdraw(double amount) {

balance -= amount;

}

}In this code, both the **accountHolder** and **balance** fields are public, allowing them to be directly accessed and modified from outside the class. While this might seem convenient, it exposes the object's internal state to potential misuse and vulnerabilities:

public class Main {

public static void main(String[] args) {

BankAccount account = new BankAccount();

account.accountHolder = "Alice";

account.balance = 1000.0;

// Malicious code

account.balance = -100000.0; // Oops, negative balance!

}

}In this example, a malicious user can directly set the **balance** to a negative value, violating the intended constraints of the banking system. This is a vulnerability that can lead to incorrect data states and undermine the security of the application.

If proper encapsulation is used, this vulnerability can be mitigated:

public class BankAccount {

private String accountHolder;

private double balance;

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

}

}

public void withdraw(double amount) {

if (amount > 0 && amount <= balance) {

balance -= amount;

}

}

// Getters and setters provide controlled access to private fields

public String getAccountHolder() {

return accountHolder;

}

public double getBalance() {

return balance;

}

public void setAccountHolder(String accountHolder) {

this.accountHolder = accountHolder;

}

}.accountHolder = accountHolder; } }

By encapsulating the **accountHolder** and **balance** fields and using proper validation checks within methods, you prevent direct manipulation of the internal state. This helps ensure that the object's data remains consistent and that unintended vulnerabilities are avoided.

but the manipulation can be done by setting the deposit into high number

You're correct that in the example I provided, an attacker could potentially manipulate the balance by depositing a large amount. However, I'd like to clarify that encapsulation itself doesn't necessarily prevent all forms of vulnerabilities. Encapsulation helps mitigate vulnerabilities by controlling access to data and providing a structured way to interact with objects, but it might not cover all possible misuse scenarios.

In the example I showed, encapsulation helps ensure that the **balance** cannot be directly set to a negative value or directly modified by external code. However, it doesn't address cases where an attacker intentionally performs actions like depositing a large amount to manipulate the balance. To address such vulnerabilities, you would need additional security measures and validation checks in place.

For instance, you might want to add further validation checks within the **deposit** and **withdraw** methods to prevent excessively large deposits or withdrawals that could lead to an unintended imbalance:

public void deposit(double amount) {

if (amount > 0 && amount <= MAX\_DEPOSIT\_AMOUNT) {

balance += amount;

}

}

public void withdraw(double amount) {

if (amount > 0 && amount <= balance && amount <= MAX\_WITHDRAW\_AMOUNT) {

balance -= amount;

}

}

In real-world applications, addressing security vulnerabilities requires a multi-layered approach that involves not only encapsulation but also input validation, authentication, authorization, and other security practices.

The primary point is that encapsulation is an important aspect of software design, but it's not the sole solution to all security-related concerns. It's just one part of building a secure and robust software system.