**Java Documentation**

**Java Environment**

* JDK = Java Development Kit 🡪 needed when developing Java program
* JRE = Java Runtime Environment 🡪 needed to run compiled Java program, usually in production server. JRE usually included when installing JDK
* JVM = Java Virtual Machine 🡪 needed to execute binary file of Java, as well as other languages (Kotlin, Scala, Groovy)

Diagram

Description automatically generated

* .java 🡪 when you make code using any IDE, name the file file.java
* .class 🡪 file.java is compiled by JDK, becomes a binary file named file.class. This file can be run using JRE
* .jar 🡪 distribution file (like using zip) for many .class files. This file can be run in JRE

**Installing Java (JDK)**

1. Download SDK from OpenJDK:

<https://openjdk.java.net/>

extract the file, not install

1. Setting environment variable dan path:

In windows:

JAVA\_HOME 🡪 directory of the extracted Java

add Path 🡪 %JAVA\_HOME%\bin

In Linux. Add to .bashrc or .profile or .zshrc:

export JAVA\_HOME=”directory of the extracted Java”

export PATH=”$JAVA\_HOME/bin:$PATH”

1. Check Java:

java -verson

javac -version

**Primitive data type in Java**

Primitive data types, have default values:

1. byte 🡪 integer, 1 byte (-128 – 127), default 0
2. short 🡪 integer, 2 bytes (-32768 – 32767), default 0
3. int 🡪 integer, 4 bytes (-2B – 2B), default 0
4. long 🡪 integer, 8 bytes (-9KT – 9 KT), default 0
5. float 🡪 floating point, 4 bytes, default 0.0
6. double 🡪 floating point, 8 bytes, default 0.0
7. Automatic conversion: byte 🡪 short 🡪 int 🡪 long 🡪 float 🡪 double
8. Manual conversion: double 🡪 float 🡪 long 🡪 int 🡪 char 🡪 short 🡪 byte
9. boolean 🡪 true or false, default value false
10. char 🡪 character, start and end with ‘ ’
11. final type 🡪 like constant
12. null 🡪 empty value, can be assigned to pointer, func data type, slice, map, channel, interface

Not primitive data types, default value is null, has methods, started with capital letter:

1. String 🡪 string, start and end with “ ”
2. Byte, Short, Integer, Long, Float, Double, Character, Boolean

Declaring variable:

byte iniByte = 100;

short iniShort = 1000;

int iniInt = 10000000;

long iniLong = 10000000000L;

float iniFloat = 10.12F;

double iniDouble = 10.12;

int iniHexa = 0xFFFF;

int iniBinary = 0b010101;

long balance = 1\_000\_000\_000\_000L; 🡪 underscore for readability

double iniDouble = iniFloat; 🡪 automatic conversion

float iniFloat = (float) iniDouble; 🡪 manual conversion, but beware of overflow

boolean iniBool = true;

char e = ‘E’;

String firstName = “Agus”;

String fullName = firstName + “ “ + lastName;

final String fixedWord = “Constant” 🡪 this variable cannot be changed

Declaring with var, it can detect the data type automatically:

var iniAngka = 8; 🡪 value must be assigned if using var

var iniHuruf = ‘a’; 🡪 value must be assigned if using var

**Print in Java**

System.out.printf("%d %3.3f %t \n", nonDecimalNum, decimalNum, boolVal)

%d 🡪 non decimal

%f 🡪 decimal

%t 🡪 bool

\n 🡪 new line

\t 🡪 tab space

**Array in Java**

Same data type, predefined length, length is fixed

Declaring array:

1. Method 1:

String[] arrayString = new String[3];

arrayString[0] = ”Susilo”;

arrayString[1] = ”Bambang”;

arrayString[2] = ”Yudhoyono”;

1. Method 2, directly assigning the values:

int[] arrayInt = new int[]{

1, 2, 3, 4, 5

};

arrayString[0] = ”Susilo”;

arrayString[1] = ”Bambang”;

arrayString[2] = ”Yudhoyono”;

1. Method 3, directly assigning the values without new:

long[] arrayLong = {

1, 2, 3, 4, 5

};

1. Array in array:

String[][] members = {

{“Eko”, “Kurniawan”},

{“Agus”, “Budi”},

};

String[] member1 = members[0];

String lastName1 = members[0][1];

Operation in array:

1. arrayName.length 🡪 taking the array’s length

**Expression, Statement, Block in Java**

Expression 🡪 construction of variable, method, code etc that resulting a value

int value = 10;

Statement 🡪 A complete execution, ended by semicolon ;

int value = 10; // an expression that is also a statement

Block 🡪 a group of statement, started and ended by bracket {}

{ int value = 10; }

**Conditional in Java**

1. if-else if-else conditionals:

if (var1 == 1) {

System.out.println("1");

} else if (var1 == 2) {

System.out.println("2");

} else {

System.out.println("other");

}

1. switch:

var nilai = “A”;

switch(nilai) {

case ”A”:

System.out.println("great");

break;

case “B”:

System.out.println("sufficient")

break;

default:

System.out.println("retake");

}

1. switch lambda, without using break, only in Java 14 and above:

var nilai = “A”;

switch(nilai) {

case ”A” -> System.out.println("great");

case “B”, “C” -> System.out.println("sufficient");

default -> { System.out.println("retake") };

}

1. switch with yield, yield is kindof return value, only in Java 14 and above:

var nilai = “A”;

String ucapan = switch(nilai) {

case ”A”:

yield “great”;

case “B”:

yield “sufficient”;

default:

yield “retake”;

}

1. Ternary operator:

var nilai = 80;

String ucapan = nilai >= 75 ? “great” : “retake”;

**Looping in Java**

1. For loop:

for (var counter = 1; counter <= 5; counter++) {

System.out.println("counter");

}

1. While loop:

var counter = 1;

while (counter <= 5) {

System.out.println("counter");

counter++;

}

1. Do While loop, at least operated once:

var counter = 1;

do {

System.out.println("counter");

counter++;

} while (counter <= 5);

1. For Each loop, only for array:

int[] arrayInt = { 1, 2, 3, 4, 5 };

for (var value : arrayInt) {

System.out.println(value);

}

Break 🡪 to totally stop the loop

Continue 🡪 To stop the current iteration, then continue to the next iteration

**Method/Function in Java**

1. Making a method/function, use static void, name with camelCase, then call it in main function:

static void sayHelloWorld() {

System.out.println("Hello World");

}

public static void main(String[] args) {

satHelloWorld();

}

1. Making a method with parameter/argument:

static void sayHelloWorld(String firstName, String lastName) {

System.out.println("Hello " + firstName + “ “ + lastName);

}

public static void main(String[] args) {

satHelloWorld(“Agus”, “Budi”);

}

1. Making a method with return value, change void with the data type of the return value:

static int sum(int angka1, int angka2) {

return angka1 + angka2;

}

public static void main(String[] args) {

var angkaTotal = sum(1, 2);

System.out.println(angkaTotal);

}

1. Method variable argument, to simplify using array as argument:

static int totalNilai(String name, int… values) {

var total = 0;

for (var value : values) {

total += value;

}

System.out.printf(“Halo %s, nilai anda %d”, nama, total);

}

public static void main(String[] args) {

// int[] values = {1, 2, 3, 4}; 🡪 don’t have to use this with variable argument

totalNilai(name: “Eko”, …values: 1, 2, 3, 4);

}

1. Method overloading 🡪 make other methods with same name, but different parameter. Overloading happens only in the same class:

static void sayHello() {

System.out.println("Hello");

}

static void sayHello(String name) {

System.out.println("Hello " + name);

}

static void sayHello(String firstName, String lastName) {

System.out.println("Hello " + firstName + “ “ + lastName);

}

public static void main(String[] args) {

sayHello();

sayHello(“Eko”);

sayHello(“Eko”, “Kurniawan”);

}

1. Recursive method 🡪 beware of stackOverflow error if the recursive stack is too deep

static int factorial(int angka) {

if (angka == 1) {

return 1;

} else {

Return angka \* factorial(angka – 1);

}

public static void main(String[] args) {

factorial(5);

}

**Scope in Java**

A variable can only be accessed in its scope, for example a variable is declared in an if block, it cannot be accessed outside this if block:

static void sayHello(String name) {

String hello = “Hello” + name;

if (!name.isBlank()) {

String hi = “Hi” + name;

}

System.out.println(hello); // normal

System.out.println(hi); // ERROR!

}

**OOP in Java**

Object 🡪 a data consists of properties/attributes/fields and methods/functions. All non primitive data type in Java are objects (String, Boolean, etc)

Class 🡪 a blueprint of object, so an object is an instance of class

* Property 🡪 attributes of class
* Method 🡪 functions in class
* Constructor 🡪 method that will be run first when the object is created. Contructor’s name must be the same as the class’ name, without void and without return value. If using constructor, when instantiating an object, the properties’ parameter must be inputted.
* Constructor overloading 🡪 possible, as long as the parameters are different
* Calling other constructors 🡪 use this
* Variable shadowing 🡪 variable is overwritten because the names are the same but the scope is different. Solution: use paramVar as name for method’s input parameter, or use this keyword
* This keyword 🡪 the current object instance that is being accessed
* Inheritance 🡪 by a child class to parent class, all properties and methods will be inherited
* Object is the parent of all child class in Java. It has many method like toString(), equals(), etc.
* Method overriding 🡪 redeclare the method in child class that overwrite the parent’s method. In which parameter must be the same.
* Super keyword 🡪 to access parent’ class’ field/property and method/function
* Super constructor 🡪 default constructor = constructor in parent class that has no parameter and body. If parent class does not have default constructor, then child class must take any existing parents’ contructor (with its parents’ contructor’s parameters).
* Polymorphism 🡪 first, instantiating an object as a parent class (Person), later on it can be transformed as its child class (Student, Teacher, etc), and every time a method is called it will return the corresponding class’ method action
* Variable hiding 🡪 when the child’s variable property has same name with parent’s property. Unlike method that can overrode, property will make a problem of variable hiding. There is no variable/property overriding. Solution: always use super.varName when calling parent’s variable
* Abstract class 🡪 class that cannot be instantiated as object, it is made as parent of a child class. Make object from this child class instead.
* Abstract method 🡪 method in a parent class to be overridden by a child class. To make sure that every child class made the method. In parent, the method must be made public.
* Encapsulation 🡪 Make sure that sensitive data in object is private, by making private all class properties. To access them, make Getter and Setter method:
  + Boolean 🡪 getter: isActive(), setter: setActive(boolean value)
  + Primitive 🡪 getter: getVar(), setter: setVar(primitive value)
  + Object 🡪 getter: getVar(), setter: setVar(object value)
* Interface 🡪 like abstract, interface is kindof contract that must be followed by the child class. Interface only consists of (empty) method and constant property/field. In child class, use implements instead of extends. So every child class override every interface’s methods.
* Interface inheritance 🡪 a class can implements many interfaces, and an interface can extends (use extends keyword) another interface:
  + Interface Car extends HasBrand
  + class Avanza implements Car, IsMaintenance 🡪 all methods must be overridden
* Default method (Java 8 and above) 🡪 the problem with empty abstract method in interface is if a new method added, every child class must override the new method. With default method, the method can be filled with block function in the interface, and every child class copy that.
  + default Boolean isBig() { return false }; 🡪 in interface body
* Object.method():
  + ToString 🡪 to make object as string, for good readability. Has default method in Object.
    - Default toString() in Object class 🡪 className + @ + hashCode
    - Can be overridden, e.g. 🡪 String toString() { return “Variable is “ + this.field1 };
  + Equals 🡪 in Java, == can only be used to compare primitive type. If not primitive (object), method Object.equals() must be used. To be safe, override the equals() method corresponding to the class’ properties, can use IDE’s generator.
  + HashCode 🡪 To be safe, override the Object.hashCode() also. just use IDE’s generator.
* Final class 🡪 class that cannot be extended by child class
* Final method 🡪 method that can be overridden by child class
* Inner class 🡪 class inside class, e.g. class Employee inside class Company. Inner class can read every private field and method of outer class, by using Company.this.field1. In main, instantiate process is like this:

Company company = new Company();

Company.Employee employee = company.new Employee();

* Anonymous class 🡪 declaring class while also instantiating its object instance. E.g.: make an interface/abstract class then make the instance of that interface, without making the class. The fast process is the advantage, but the downside is it cannot be reused, commonly used for simple task. In main, type:

Interface1 object1 = new Interface1() { //fill in the methods };

* Static keyword 🡪 make field, method, inner class, or block can be accesses without through its object, or without having to make the class instance. Static variables commonly written all in CAPITAL.
* Static import 🡪 import static packageName 🡪 so in the body we don’t have to write className.var1 but instead writing var1 directly
* Record class 🡪 class commonly used only to contain immutable/final data, automatically will make getter, equals, hashCode, toString, constructors (input that will be the class fields)
* Enum class 🡪 class/data type with limited/enum value, like making a struct data type.
* Exception/Error 🡪 error is reckoned as class, extended from parent class Throwable. In other class X, if there is error call the error class using throws after class name and keyword throw in method body. In main class, calling class X is possible to make error and the IDE usually will hint you. To be safe, use try-catch(-finally) expression. Type of exception:
  + Checked exception 🡪 have to be collected by try-catch, the one explained above
  + Runtime exception🡪 does not have to be collected by try-catch: NullPointerException, IllegalArgumentException, etc 🡪 but better to try-catch
  + Error 🡪 fatal problem, like database connection fail 🡪 not recommended to try-catch, just stop the program!
* StackTraceElement class 🡪 method in throwable to show where the error happens
* Try with resource (Java 7 and above) 🡪 in try block, if using resource it must be closed at the end using interface AutoCloseable, e.g. when reading files
* Annotation 🡪 giving metadata to program, usually when making library. Can be accessed using Reflection. To make annotation, use keyword @interface.
  + @Target 🡪 to inform this annotation can be use in which class/method/field/etc?
  + @Retention 🡪 to determine if annotation will be kept in compiled result or not
* Java predefined annotation:
  + @Override 🡪 method overrides parent’s method
  + @Deprecated 🡪 method is not recommended to use
  + etc
* Reflection 🡪 To view the structure of our apps, useful when making library/framework. To access reflection, use method getClass() or ClassName.class

Declaring a class, make a new file:

class Person {

String name;

String address;

Person(String paramName, String paramAddress) {

this.name = paramName;

this.address = paramAddress;

}

Person(String paramName) { // first overloading, calling first constructor

this(paramName, null);

}

Person() { // second overloading, calling second constructor

this(null);

}

void sayHello(String paramCaller) {

System.out.println(“Hello “ + paramCaller + “, my name is “ + this.name);

}

}

Making an inheriting child class from parent child:

class Student extends Person {

Student(String paramName, String paramAddress) {

super(paramName, paramAddress); // super constructor

}

void sayHello(String paramCaller) { // method overriding

System.out.println(“Hello “ + paramCaller + “, this is student “ + this.name);

}

void sayHelloParent(String paramCaller) { // super method

super.sayHello(paramCaller);

}

}

Making another inheriting child class to show polymorphism:

class Teacher extends Person {

Teacher (String paramName, String paramAddress) {

super(paramName, paramAddress); // super constructor

}

void sayHello(String paramCaller) { // method overriding

System.out.println(“Hello “ + paramCaller + “, this is teacher “ + this.name);

}

}

Instantiating an object, call it in main file:

var person1 = new Person(“Eko”, “Jakarta”);

person1.sayHello(“Asep”); // “Hello Asep, my name is Eko”

var student1 = new Student(“Budi”, “Bandung”);

student1.sayHello(“Asep”); // “Hello Asep, this is student Budi”

student1. sayHelloParent (“Asep”); // “Hello Asep, my name is Budi”

var person = new Person("Abdul", "Bogor"); // variable person of Person class in Polymorphism

person = new Student("Abdul", " Bogor "); // transform person to Student class

person.sayHello("Asep"); // “Hello Asep, this is **student** Abdul”

person = new Teacher("Abdul", " Bogor "); // transform person to Teacher class

person.sayHello("Asep"); // “Hello Asep, this is **teacher** Abdul”

**Type checking in Java**

Use instanceof:

(person1 instanceof Person) 🡪 return Boolean value

**Casting class in Java**

From child class to parent class:

Student student2 = new Student(“Abdul”, “Bogor”);

Person person2 = (Person) student2; // beware of variable hiding problem in this case

**Package in Java**

To gather many classes/files inside one directory/folder

To make a package (1 folder = 1 package):

1. Right click at src folder 🡪 new 🡪 package, name the package, folder inside need written with dot . e.g. lib.data and lib.app (there will be folder lib/data and lib/app)
2. In every class file inside this folder, type at the first line:

package lib.data;

To import a package from another class file from different package (1 import = 1 class file). The imported class must be public:

1. In a class file at which you want to import another class file:

import lib.data.table;

1. To import all class in a folder:

import lib.data.\*;

**Access modifier in Java**

To determine which class, field/property, method, and constructor can be accessed by which actor

* Public 🡪 can be accessed by same: class, package, subclass, world. Only 1 public class in 1 file, class name must be the same with file name.
* Protected 🡪 can be accessed by same: class, package, subclass
* No modifier (blank) 🡪 can be accessed by same: class, package
* Private 🡪 can be accessed by same: class