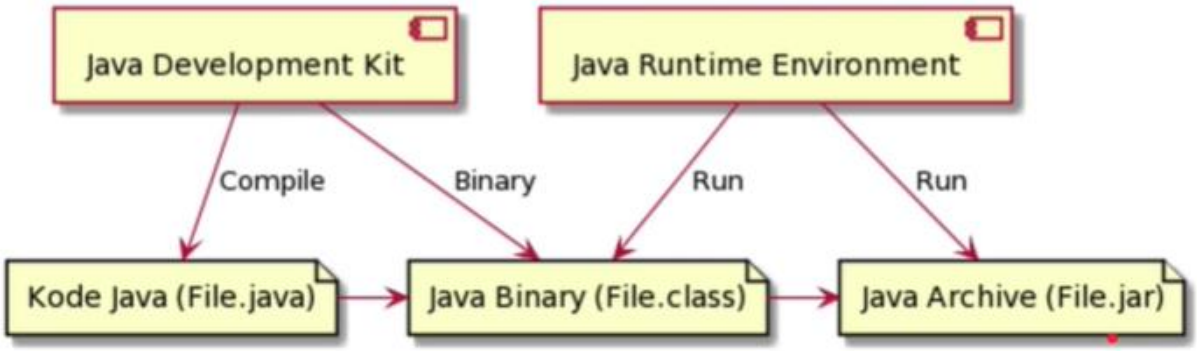
Java Documentation

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)



- .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
2. 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”
3. Check Java:
   java -verson
   javac -version

Other method using oracle Java in Mac:

1. Download SDK from first link google “oracle java sdk download”:
2. Choose the file: x64 DMG Installer jdk-18_macos-x64_bin.dmg, download it
3. In downloaded file, double click, install it
4. Download IntelliJ .dmg dile, install it by drag drop to app

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";
2. Method 2, directly assigning the values:
   int[] arrayInt = new int[]{
   1, 2, 3, 4, 5
   };
   arrayString[0] = "Susilo";
   arrayString[1] = "Bambang";
   arrayString[2] = "Yudhoyono";
3. Method 3, directly assigning the values without new:
   long[] arrayLong = {
   1, 2, 3, 4, 5
   };
4. 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");
}
```

2. 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");
}
```

3. 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") };
```

```
}
```

4. 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”;
}
```

5. 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");
}
```

2. While loop:

```
var counter = 1;
while (counter <= 5) {
   System.out.println("counter");
   counter++;
}
```

3. Do While loop, at least operated once:

```
var counter = 1;
do {
   System.out.println("counter");
   counter++;
} while (counter <= 5);
```

4. 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();
}
```

2. 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");
}
```

3. 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);
}
```

4. 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);
}
```

5. 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");
}
```

6. 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;

2. 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

Standard Class in Java

1. String class
   Immutable, to modify String actually we make new String
   String method:
   String text.toLowerCase() → change the text to lower case
   String text.toUpperCase() → change the text to upper case
   int text.length() → get the text's length
   boolean text.startsWith(value) → check if the text started with the value
   boolean text.endsWith(value) → check if the text ended with the value
   String[] text.split(value) → split the text based on the value as delimitor
   boolean text.isBlack() → check if text contains no any valid char
   boolean text.isEmpty() → check if text is null
   StringBuilder → fast but not thread-safe
   StringBuilder text = new StringBuilder();
   text.append("Eko");
   text.append("Kurniawan");
   String fullname = text.toString(); // return "EkoKurniawan"
   StringJoiner → join several String with delimiter, prefix, and suffix
   StringJoiner text = new StringJoiner();
   text.add("Eko");
   text.add("Kurniawan");
   String fullname = text.toString(", ", "{", "}"); // return "{Eko, Kurniawan}"
   StringTokenizer → split String based on delimiter, lazy characteristic, useful for large text
   String fullname = " Eko Kurniawan";
   StringTokenizer tokenizer = new StringTokenizer (fullname, " ");
   While (tokenizer.hasMoreTokens()) {
   String token = tokenizer.nextToken(); // return "Eko" then "Kurniawan"
   }

2. Number class
   It is the parent of all number class
   Number method:
      num.intValue(), num.floatValue(), num.byteValue(), etc → change the type
   Converting String to Number:
      parseLong(text), parseInteger(text), parseShort(text), parseByte(text) → change to primitive
      valueOf(text) → change to non primitive number data type
3. Math class
   Mathematics **static** methods in Java, doesn't have to make the object instance
      double Math.sqrt(num) → square root of num
      double Math.powerofTwoD(num) → num*num
      double Math.log10(num) → log(num)
      Math.ceil(num), Math.floor(num), Math.max(num), Math.abs(num), Math.PI, etc
4. BigInteger, BigDecimal
   To provide data types that exceed maximum capacity of Long and Double
5. Scanner class
   To read input from console, file, etc
      String scanner.nextLine() → read string data
      int scanner.nextInt() → read int data
      long scanner.nextLong() → read long data
      boolean scanner.nextBoolean() → read boolean data
   Example:
      Scanner scanner = new Scanner.(System.in); //System.in is how to read console input
      String nama = scanner.nextLine();
6. Calendar class and Date class
   For making a datetime variable. Date class' methods are deprecated, only use its date.getTime() only
   Example
      Date tgl = new Date(23587200000L); // input the millisecond, return Thu Oct 01 07:00:00 1970
      Calendar calendar = new Calendar.getInstance();
      Calendar.set(Calendar.YEAR, 2000);
      Calendar.set(Calendar.MONTH, Calendar.JANUARY);
      Calendar.set(Calendar.DAY_OF_MONTH, 17);
      Calendar.set(Calendar.HOUR_OF_DAY, 10);
      Calendar.set(Calendar.MINUTE, 0);
      Calendar.set(Calendar.SECOND, 0);
      Calendar.set(Calendar.MILLISECOND, 0);
      Date date = calendar.getTime();
      Long millisecond = date.getTime(); // return Mon Jan 17 10:00:00 2000
7. System class
   Many utility **static** methods in Java, doesn't have to make the object instance
      System.getenv() → get environment variables
      System.currentTimeMilis() → get the current timestamp
8. Runtime class
   To get the information of environment the Java run in

Runtime runtime = Runtime.getRunime();
Runtime.Processors(); → get numbers of core cpu
Runtime freeMomory(); → number of free memory in JVM

9. Objects class
   Many utility static method used to check before running operation, safe from null input
   Objects.toString(data) → still okay if the data is null
   System.hashCode(data) → still okay if the data is null
   System.equals(data1, data2) → still okay if the data is null
10. Random class
    To generate a random value
    random.nextInt(maxVal) → generate a random int below 1000
11. Properties class
    Class to store only property, key=value pairs, e.g. to store env variable
12. Arrays class
    Many static methods for array
    Arrays.binarySearch(array1, value) → search value in an ordered array
    Int[] newArr = Arrays.copyOf(array1, startIndex, endIndex) → copy the contents of array
    Arrays.equals(array1, array2) → return boolean is arrays equal
    Arrat.sort(array1) → to order array
    Arrays.toString(array1) → change array to String
13. RegEx search

```
String name = "Eko Kurniawan Khannedy Programmer Zaman Now";
Pattern pattern = Pattern.compile("[a-zA-Z]*[a][a-zA-Z]*"); // Starts-ends with any, 'a' in mid
Matcher matcher = pattern.matcher(name);
While (matcher.find()){
   String result = matcher.group(); // returns Kurniawan, Khannedy, Programmer, Zaman
}
```

Java Generic Class

Ability to add generic parameter type T/E/K/N/V/etc, so we can change the data type along the way.

In normal way: casting with

```
Object intValue = 1
String stringValue = (String) intValue → success at compiler, error at runtime
```

Using generic type:

```
T intValue = 1
String stringValue = (String) intValue → error at compiler already
```

Declaring:

```
public class MyData<T, U> {
  private T data1;
```

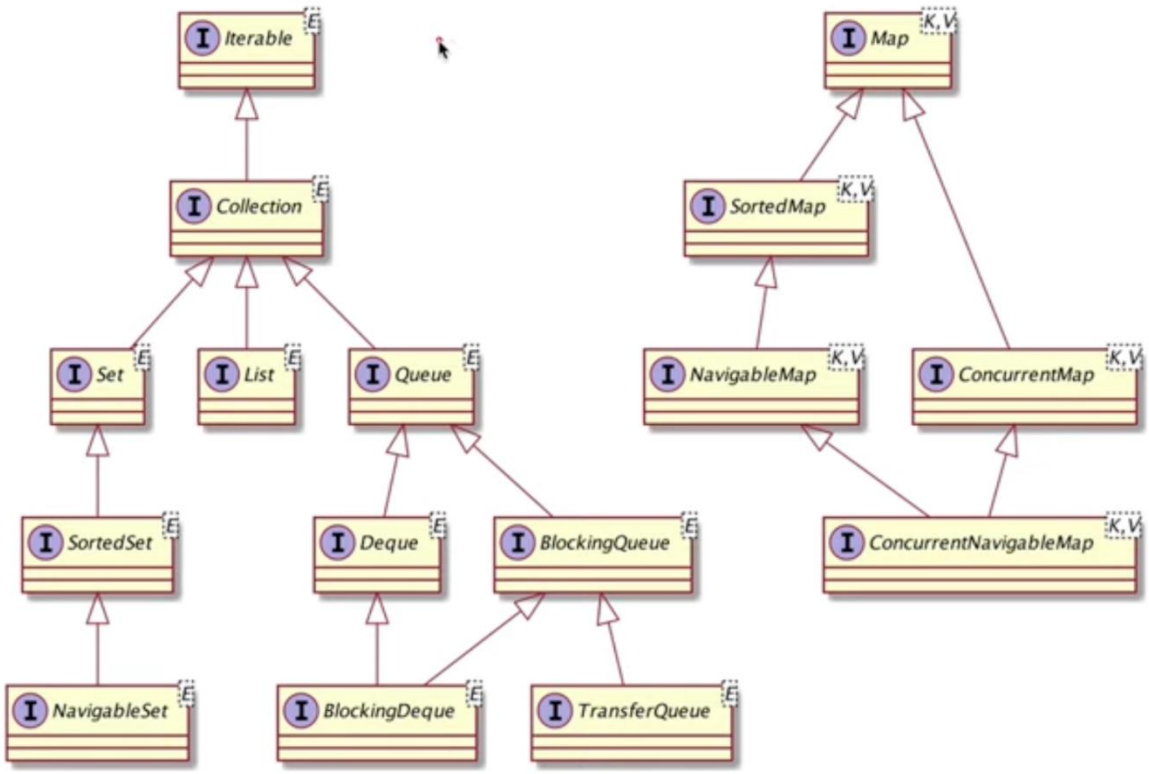
```
    private U data2

    public MyData(T data1, U data2) {
      this.data1 = data1;
      this.data2 = data2;
    }

    public T getData1() {
      return data1;
    }

    public void setData1(T data1) {
      this.data = data1;
    }

    public static <U> countLength(U[] array1){
      return array1.length;
    }
  }
```

At main class:

```
MyData<String, Integer> myData1 = new MyData<String>(”Eko”, 10);
MyData<Integer, String> myData2 = new MyData<String>(10, “Eko”);
```

Collections at Java

For data structure making. All collections have interface contract of method and its implementation



1. Iterable interface
   Parent of all collections except map, contract to iterate, support for-each loop
   ```
   Iterable<String> names = List.of("Eko", "Kurniawan", "Khannedy");
   for (var name : names) {
     // do something;
   }
   ```
2. Collection interface
   Contract to manipulate collection data (add, delete, peek, etc)
   No direct implementation, only as a parent for List, Set, Queue
   Methods:
   collection.size() → returns the size of a collection
   collection.isEmpty() → check if the collection empty
   collection.contains(value) → check if in collection include element E
   collection.add(value) → add element E to collection
   collection.remove(value) → remove element E from collection
3. List interface
   Can contain duplicate values, in order of inputting the data, has index, like array but dynamic size
   Methods (addition, includes Collections' methods):
   list.get(index) → get the value at index
   list.set(index, value) → get the value at index

list.indexOf(value) → return the index of value at list
list.sort() → sort the list

Type:

ArrayList → array with default size 10, if there more element then make new array, so dynamic
LinkedList → node and pointer to next and previous node

Operasi	ArrayList	LinkedList
add	Cepat jika kapasitas Array masih cukup, lambat jika sudah penuh	Cepat karena hanya menambah node di akhir
get	Cepat karena tinggal gunakan index array	Lambat karena harus di cek dari node awal sampai ketemu index nya
set	Cepat karena tinggal gunakan index array	Lambat karena harus di cek dari node awal sampai ketemu
remove	Lambat karena harus menggeser data di belakang yang dihapus	Cepat karena tinggal ubah prev dan next di node sebelah yang dihapus

Declaration:

```
List<String> names = new ArrayList<>(100); //100 is initial capacity, default is 10
// List<String> names = new LinkedList<>();
names.add("Eko");
names.add("Khannedy");
names.set(0, "Budi"); //change index 0 to "Budi"
names,remove(1); // remove index 1
names.get(0); // get index 0
```

4. Immutable List

Data is final, cannot be changed, use when the data should not be arbitrarily changed

Methods:

Collections.emptyList() → create immutable empty list
Collections.singletonList(e) → create immutable list with 1 element
Collections.unmodifiableList(list) → convert mutable list becomes immutable
List.of(e1, e2, e3) → create immutable list from elements

5. Set interface

Cannot contain duplicate values, no index, no new method (only from Collection and Iterable)
No index means to get elements we have iterate one-by-one

Type:

HashSet → not ensure that data ordered as the order of inputting data
LinkedHashSet → ensure that data ordered as the order of inputting data

Declaration:

```
Set<String> names = new HashSet<>();
// Set<String> names = new LinkedHashSet<>();
names.add("Eko");
names.add("Khannedy");
for (var name : names) {
   // do something
}
```

6. Immutable Set
   Data is final, cannot be changed, use when the data should not be arbitrarily changed
   Methods:
   - Collections.emptySet() → create immutable empty set
   - Collections.singletonSet(e) → create immutable set with 1 element
   - Collections.unmodifiableSet(list) → convert mutable set becomes immutable
   - Set.of(e1, e2, e3) → create immutable set from elements
7. Sorted Set interface
   The elements will be ordered by value (ascending/descending), not by the order of inputting
   Methods:
   - sortedSet.subSet(EStart, Eend) → get the element from Estart to Eend
   - sortedSet.headSet(E) → get the first element until element E
   - sortedSet.tailSet(E) → get the element E until the last element
   - sortedSet.first() → get the first element
   - sortedSet.last() → get the last element

   Declaration: use **TreeSet**
   - SortedSet<String> names = new TreeSet<>();
   - names.add("Eko");
   - names.add("Kurniawan");
8. Immutable Sorted Set
   Methods:
   - Collections.emptySortedSet() → create immutable empty sorted set
   - Collections.unmodifiableSortedSet() → convert mutable sorted set becomes immutable
9. NavigableSet interface
   Child of sorted set, can use its methods, added by methods for comparing values, etc
   Methods:
   - navigableSet.lower(E) → get the elements smaller than E
   - navigableSet.descendingSet() → reverse the navitableSet
   - navigableSet.subSet(E1, bool1, E2, bool2) → get element from E1 (include/not) to E2 (incld/not)

   Declaration:
   - NavigableSet<String> names = new TreeSet<>();
   - names.addAll(Set.of("A", "B", "C", "D"));
   - NavigableSet<String> namesReverse = names.descendingSet(); // D C B A
   - NavigableSet<String> subNames = names.subSet("A", true, "B", true); // A B
10. Immutable Navigable Sorted Set
    Methods:
    - Collections.emptyNavigableSet() → create immutable empty navigable set
    - Collections.unmodifiableNavigableSet() → convert mutable navigable set becomes immutable
11. Queue interface
    FIFO
    Methods:
    - queue.add(E) → collections' method, add as the last element, if error return exception
    - queue.offer(E) → same with add, but if error return false
    - queue.remove() → remove the first element, if no element return NoSuchElementException

queue.poll() → remove the first element, if no element return null
queue.element() → get the first element without removing, if empty return exception
queue.peek() → get the first element without removing, if empty return null

Types

ArrayDeque → built on array, has maximum seize in default
LinkedList → built on LinkedList, no maximum size in default
PriorityQueue → there is priority, ascending data, e.g. for sorting importance

Declaring

```
Queue<String> queue = new ArrayDeque<>(10); // default size is 10, but can grow itself
queue.offer("Eko");
queue.offer("Kurniawan");
for (String next = queue.poll(); next != null; next = queue.poll()) {
  // do something
}
```

12. Deque interface

Double ended queue, can operate at forward or backward, LIFO and FIFO, can be for stack

Methods:

deque.addFirst(E) → add as the first element
deque.addLast(E) → add as the last element
offerFirst/Last(E), removeFirs/Lastt(), pollFirst/Last(), getFirst/Last(), peekFirst/Last()
deque.push(E) → push as the last elemet
deque.pop(E) → pop/remove the first elemet, return that element

Types

ArrayDeque → built on array, has maximum seize in default
LinkedList → built on LinkedList, no maximum size in default

Declaring stack with deque

```
Queue<String> stack = new LinkedList<>();
stack.offerLast("Eko");
stack.offerLast ("Kurniawan");
for (String next = stack.pollLast(); next != null; next = stack.pollLast()) {
  // do something
}
```

Declaring queue with deque

```
Queue<String> queue = new LinkedList<>();
stack.offerLast("Eko");
stack.offerLast ("Kurniawan");
for (String next = stack.pollFirst(); next != null; next = stack. pollFirst()) {
  // do something
}
```

13. Map interface

Key-Value pair, no duplicate key (if duplicate mean overwrite value)

Methods:

map.size() → return the map's size
map.isEmpty() → check if map empty

map.containsKey(key) → check if map contains key
map.containsValue(value) → check if map contains value
map.get(key) → get the value of a key
map.put(key, value) → add a new key-value pair (or update for existing key)
map.remove(key) → remove the key-value pair, return the value
map.values() → returns Collection, for iteration

Types

HashMap → map that distributes its key based on hashCode() function
WeakHashMap → if not used anymore, in next garbage collection (System.gc()), key is deleted
IdentityHashMap → data are equal if they are really in a same memory (use == as equal)
LinkedHashMap → map using double linked list, inputting order is the map order
EnumHashMap → only enum type key

Declaring

```
Map<String, String> map = new HashMap<>();
// Map<String, String> map = new WeakHashMap<>();
map.put("firstName", "Eko");
map.put("lastName", "Kurniawan");
map.get("firstName"); // return Eko
map.forEach(new BiConsumer<String, String>() { // iterate to key and value
  @Override
  public void accept(String key, String value) {
    // do something
  }
}
map.forEach((key, value) -> System.out.println(key + ":" + value)); // iterate with lambda func
```

14. Immutable Map

Methods:

Collections.emptyMap() → create immutable empty map
Collections.unmodifiableMap(map) → convert mutable map becomes immutable
Collections.singleton(key, value) → make immutable map with single key-value pair
Map.of(key1, value1, key2, value2) → make immutable map with key-value pairs

15. SortedMap Interface

Map that is ordered by comparable key

Methods:

.comparator() → create comparator, for for-loop
subMap(K1, K2) → get sub sorted map from K1 to K2
headMap(K) → get sub sorted map from the first to K key
tailMap(K) → get sub sorted map from K key to the last key
firstKey() → get the first key
lastKey() → get the lastkey

Declaring: use **TreeMap**

```
SortedMap<String, String> map = new TreeMap<>(comparator);
map.put("Eko", "Eko");
map.put("Budi", "Budi");
```

```
for (var key : map.keySet()) {
  // do something
}
```

16. Immutable SortedMap
    Methods:
    - Collections.emptySortedMap() → create immutable empty sorted map
    - Collections.unmodifiableSortedMap(map) → convert mutable sorted map becomes immutable
17. NavigableMap interface
    Child of sorted map, can use its methods, added by methods for comparing values, etc
    Methods:
    - navigableMap.lowerEntry(E) → get the keys-values smaller than E
    - navigableMap.lowerKey(E) → get the keys smaller than E
    - navigableMap.descendingMap() → reverse the navigable map
    - navigableMap.subMap(E1, bool1, E2, bool2) → get key-value from E1 (incl/not) to E2 (incl/not)

    Declaration:

```
NavigableMap<String> map = new TreeMap<>();
map.addAll(Map.of("A", "char A", "B", "char B"));
NavigableMap<String> mapReverse = map.descendingMap();
NavigableMap<String> subMap= map.subMap("A", true, "B", true);
```

18. Immutable Navigable Map
    Methods:
    - Collections.emptyNavigableMap() → create immutable empty navigable map
    - Collections.unmodifiableNavigableMap() → convert mutable navigable map immutable
19. Entry Map
    Entry is a simple interface, an implementation of pair (key-value) for map
    Entry has methods to get/set key and value
    Methods:
    - entry.getKey() → get the key of the entry
    - entry.getValue() → get the value of the entry
    - entry.setValue(V) → set the value of the entry with V

    Declaration:

```
Map<String, String> map = new HashMap<>();
map.put("firstName", "Eko");
map.put("lastName", "Kurniawan");
Set<Map.Entry<String, String>> entries = map.entrySet();
for (var entry : entries) {
  var key = entry.getKey();
  var value = entry.getValue();
}
```

20. Legacy Collection
    Vector, HashTable class → synchronized, safe for multithreading
    Stack class → LIFO, use conventional pop and push, not commonly used since the reign of Deque
21. Sorting
    Only at List; while Set, Queue, Deque, Map have their own mechanisms

Methods:

Collections.sort(list) → sort the list with built in comparable

Collections.sort(list, comparator) → sort the list with self-made comparator

Declaration:

```
List<String> names = new ArrayList<>();
names.addAll(List.of(”Eko”, “Budi”, “Joko”);
Collections.sort(names); // ascending sorting
Collections.sort(names, new Comparator<String>() { // descending sorting, with anonym class
  @Override
  public int compare(String o1, String o2) {
    return o2.compareTo(o1);
  }
}
Comparator<String> reverse = new Comparator<String>() {
  @Override
  public int compare(String o1, String o2) {
    return o2.compareTo(o1);
  }
}
Collections.sort(names, reverse) // descending sorting, with self-made comparator
```

22. Binary Search

Deafult search in List is sequential search, quite slow

Binary search is fast, but can only work if the data has been sorted

Java has an implementation of binary search

Methods:

Collections.binarySearch(list, value) → binary search method

Collections.binarySearch(list, value, comparator) → binary search with self-made comparator

Declaration:

```
List<Integer> numbers = new ArrayList<>();
for (int i = 0; i < 1000; i++) { numbers.add(i); }
int index = Collections.binarySearch(numbers, 500); // normal binary search
Comparator<Integer> reverse= new Comparator<Integer>() {
  @Override
  public int compare(Integer o1, Integer o2) {
    return o2.compareTo(o1);
  }
}
int index = Collections.binarySearch(numbers, 500, reverse) // inverse binary search
```

23. Collections class

Has many utility static methods:

void copy(listTo, listFrom) → copy all data from listFrom to listTo

int frequency(collection, object) → return how many element that is same with object

max(collection), max(collection,comparable) → return max element

void revere(list) → reverse the list

void shuffle(list) → shuffle the list
void swap(list, from, to) → swap position from 'from' to 'to' in list

24. Abstract Collection

Java provides the abstract class for every collection, as the base algorithm, if you want to manually create your own collection, just extend these abstracts classes:

AbstractCollection, AbstractList, AbstractMap, AbstractQueue, AbstractSet

25. Default Method

All collections in Java have interface contract, we can improve the collection method interface with default method

Default method in Collection

Iterable.forEach(consumer) → do iteration for every collection data
List.removeIf(predicate) → remove data in collection using predicate if
List.replaceAll(operator) → change all data in collection

Default method in Map

getOrDefault(key, defaultValue) → get data base on key, if no data, return defaultValue
forEach(consumer) → do iteration for every key-value pair
replaceAll(function) → change all value data
putIfAbsent(key, value) → save data to map if not exist
remove(key, value) → remove if key-value is the same

26. Spliterator Interface

To make collection partitioning that is safe for multi threading

Method

collection.spliterator() → convert collection to be a spliterator
spliterator.trySplit() → split the spliterator
spliterator.estimateSize() → estimate the size of the spliterator
spliterator.forEachRemaining() → iterate in the remaining spliterator

27. Converting to Array

Method to convert Collection to Array

Object[] toArray() → convert collection to array
T[] toArray(T[]) → convert collection to array T (type specified)

Declaration

```
List<String> names = List.of("Eko", "Kurniawan");
Object[] objects = names.toArray();
String[] strings = names.toArray(new String[]{});
```