

# Bahasa Pemrograman 2

**Desain Class dan Obyek**

anton@ukdw.ac.id

# Konstruktor

- **Konstruktor** digunakan pada saat instansiasi sebuah object.
- Melakukan instansiasi adalah mengalokasikan sejumlah **memory** dari komputer untuk sesuatu kebutuhan struktur data yang digunakan
- Konstruktor pada Java namanya harus sama dengan nama classnya
- Konstruktor bersifat **opsional**
- **super()**, bisa digunakan untuk memanggil konstruktor yang ada pada superclass

# Konstruktor

- Mempunyai karakteristik *seperti metode*, dapat mempunyai parameter dan dapat mengakses variabel anggota
- Namun ada perbedaan prinsip antara konstruktor dan metode yaitu :
  - Nama konstruktor PASTI sama dengan nama **Kelas**.
  - Konstruktor tidak pernah mempunyai tipe data, ataupun **void**
  - Konstruktor tidak bisa mengembalikan nilai, tidak boleh ada statement **return**.
  - Hanya dieksekusi (dipanggil) saat **instansiasi** saja.

## Konstruktor (2)

- Dalam suatu kelas pasti (harus) mempunyai konstruktor, jika dalam suatu kelas tidak ada satu konstruktor pun yang dibuat, maka kompiler java secara otomatis akan membuat konstruktor kosong (***null constructor***).
- Jika dalam suatu kelas sudah ada minimal satu konstruktor, maka konstruktor otomatis tidak dibuat oleh kompiler **Java**.
- Konstruktor digunakan untuk **inisialisasi** obyek!

## Konstruktor (3)

- Dalam suatu kelas dapat mempunyai konstruktor lebih dari satu, dengan syarat tiap-tiap konstruktor harus mempunyai ***signature*** yang berbeda satu dengan yang lain.
- Yang dimaksud dengan ***signature*** adalah bentuk parameter formal konstruktor tersebut, yaitu :
  - banyaknya parameter formal
  - tipe-tipe data dari tiap parameter formal
  - serta urutannya letak parameter formalnya.
- Adanya lebih dari satu konstruktor ini disebut dengan **overloading konstruktor**.

## Konstruktur (4)

- Contoh Constructor:

```
class Mobil{  
    int jumRoda;  
  
    public Mobil(int jumRoda){  
        this.jumRoda = jumRoda;  
    }  
}
```

## Konstruktur (4)

- Contoh Overloading Constructor:

```
class Mobil{
    int jumRoda;

    public Mobil(int jumRoda){
        this.jumRoda = jumRoda;
    }
    public Mobil(){
        System.out.println("mobil baru");
    }
}
```

# Kata kunci “this”

- Dipergunakan pada sebuah kelas dan digunakan untuk menyatakan obyek **sekarang** / kelas itu

```
| public class Mahasiswa{  
    String nim;  
    String nama;  
    float ipk;  
  
|     void cetakNIMdanNama () {  
        System.out.println ("NIM : " + this.nim);  
        System.out.println ("Nama : " + this.nama);  
-     }  
  
|     void setNama (int nm) {  
        this.nama = nm;  
-     }  
- }
```

# Lingkup variabel

```
public class Lingkup{
    String warna = "Merah";
    void infoLingkup(){
        String warna = "Biru";
        System.out.println ("Warna pada metode : " + warna);
        System.out.println ("Warna milik kelas : " + this.warna);
    }
    public static void main (String[] args) {
        Lingkup l = new Lingkup();
        l.infoLingkup();
    }
}
```

# Class Design Principles

- Coupling
- Cohesion
- Code Duplication

# Coupling

- **Keterikatan** antar class
- Tight coupling vs Loose coupling
- Keterikatan **lemah**: perubahan pada suatu class **tidak** memiliki pengaruh besar pada class lain

# Cohesion

- Satu unit kode hanya melakukan **satu** tugas
- Cohesion dapat diterapkan pada tingkatan **class** atau **method**
- Method cohesion: satu method hanya melakukan satu tugas saja
- Class cohesion: satu kelas hanya mewakili satu entitas tunggal

# Methods Cohesion

```
public void kursKeDollar(int rupiah) {  
    System.out.println("=====");  
    System.out.println("Kurs Dollar ke Rupiah");  
    System.out.println("=====");  
    System.out.println("Rupiah : "+rupiah);  
    System.out.println("Dollar : "+rupiah*10000);  
}
```

# Methods Cohesion

```
public void kursKeDollar(int rupiah) {  
    System.out.println("=====");  
    System.out.println("Kurs Dollar ke Rupiah");  
    System.out.println("=====");  
  
    System.out.println("Rupiah : "+rupiah);  
    System.out.println("Dollar : "+rupiah*10000);  
}
```

# Methods Cohesion

```
private void showDesc() {  
    System.out.println("=====");  
    System.out.println("Kurs Dollar ke Rupiah");  
    System.out.println("=====");  
}
```

# Methods Cohesion

```
public void kursKeDollar(int rupiah) {  
    showDesc();  
  
    System.out.println("Rupiah : "+rupiah);  
    System.out.println("Dollar : "+rupiah*10000);  
}
```

# Code Duplication

- Duplikasi kode akibat dari rancangan kelas yang **tidak baik**
- Perubahan pada suatu bagian **harus** diikuti perubahan pada bagian lain
- Proses maintenance dan pengembangan menjadi lebih sulit

# Instansiasi

- Membutuhkan operator **new** untuk mempersiapkan memory sesuai dengan isi kelas
- Berbeda dengan variabel primitif yang menyimpan data pada variabel-nya, variabel objek menunjuk **alamat** objek baru yang dibuat
- Membutuhkan konstruktor untuk membantu membentuk / menginisialisasi objek yang bersangkutan.

# Contoh instansiasi Date

```
import java.util.Date;
public class Tanggal {

    /**
     * Creates a new instance of <code>Tanggal</code>.
     */
    public Tanggal() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        Date tgl = new Date();
        System.out.println ("Tanggal sekarang adalah : " + tgl);
        System.console().readLine();
    }
}
```

# Date

## Constructor Summary

[Date](#)()

Allocates a Date object and initializes it so that it represents the time at which it was allocated, measured to the nearest millisecond.

[Date](#)(int year, int month, int date)

**Deprecated.** *As of JDK version 1.1, replaced by `Calendar.set(year + 1900, month, date)` or `GregorianCalendar(year + 1900, month, date)`.*

[Date](#)(int year, int month, int date, int hrs, int min)

**Deprecated.** *As of JDK version 1.1, replaced by `Calendar.set(year + 1900, month, date, hrs, min)` or `GregorianCalendar(year + 1900, month, date, hrs, min)`.*

[Date](#)(int year, int month, int date, int hrs, int min, int sec)

**Deprecated.** *As of JDK version 1.1, replaced by `Calendar.set(year + 1900, month, date, hrs, min, sec)` or `GregorianCalendar(year + 1900, month, date, hrs, min, sec)`.*

[Date](#)(long date)

Allocates a Date object and initializes it to represent the specified number of milliseconds since the standard base time known as "the epoch", namely January 1, 1970, 00:00:00 GMT.

[Date](#)(String s)

**Deprecated.** *As of JDK version 1.1, replaced by `DateFormat.parse(String s)`.*

## Method Summary

boolean	<a href="#">after</a> (Date when) Tests if this date is after the specified date.
boolean	<a href="#">before</a> (Date when) Tests if this date is before the specified date.
<a href="#">Object</a>	<a href="#">clone</a> () Return a copy of this object.
int	<a href="#">compareTo</a> (Date anotherDate) Compares two Dates for ordering.
boolean	<a href="#">equals</a> (Object obj) Compares two dates for equality.
int	<a href="#">getDate</a> () <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.get(Calendar.DAY_OF_MONTH)</code> .
int	<a href="#">getDay</a> () <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.get(Calendar.DAY_OF_WEEK)</code> .
int	<a href="#">getHours</a> () <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.get(Calendar.HOUR_OF_DAY)</code> .
int	<a href="#">getMinutes</a> () <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.get(Calendar.MINUTE)</code> .
int	<a href="#">getMonth</a> () <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.get(Calendar.MONTH)</code> .
int	<a href="#">getSeconds</a> () <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.get(Calendar.SECOND)</code> .

long	<a href="#">getTime()</a> Returns the number of milliseconds since January 1, 1970, 00:00:00 GMT represented by this Date object.
int	<a href="#">getTimezoneOffset()</a> <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>-(Calendar.get(Calendar.ZONE_OFFSET) + Calendar.get(Calendar.DST_OFFSET)) / (60 * 1000)</code> .
int	<a href="#">getYear()</a> <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.get(Calendar.YEAR) - 1900</code> .
int	<a href="#">hashCode()</a> Returns a hash code value for this object.
ic long	<a href="#">parse(String s)</a> <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>DateFormat.parse(String s)</code> .
void	<a href="#">setDate(int date)</a> <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.set(Calendar.DAY_OF_MONTH, int date)</code> .
void	<a href="#">setHours(int hours)</a> <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.set(Calendar.HOUR_OF_DAY, int hours)</code> .
void	<a href="#">setMinutes(int minutes)</a> <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.set(Calendar.MINUTE, int minutes)</code> .
void	<a href="#">setMonth(int month)</a> <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.set(Calendar.MONTH, int month)</code> .
void	<a href="#">setSeconds(int seconds)</a> <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.set(Calendar.SECOND, int seconds)</code> .
void	<a href="#">setTime(long time)</a> Sets this Date object to represent a point in time that is <code>time</code> milliseconds after January 1, 1970 00:00:00 GMT.
void	<a href="#">setYear(int year)</a> <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>Calendar.set(Calendar.YEAR, year + 1900)</code> .
String	<a href="#">toGMTString()</a> <b>Deprecated.</b> As of JDK version 1.1, replaced by <code>DateFormat.format(Date date), using a GMT TimeZone</code> .

# Dynamic Creation

- Objek mengalami dynamic creation:
  - Program menciptakan sejumlah objek menurut pola yang *tidak mungkin* diprediksi pada saat kompilasi
  - Pada saat operasi terjadi, objek baru mungkin saja diciptakan, reference sebuah objek diubah ke objek lain atau tidak lagi mengacu ke objek apapun
  - Nilai yang diacu dalam sebuah field objek diubah

# Penghancuran Objek

- Ada kemungkinan banyak “**sampah**” obyek!
- Teknik yang digunakan java untuk menangani objek yang sudah tidak diperlukan lagi disebut **garbage collection**.
- Objek yang sudah tidak diperlukan lagi akan terdeteksi oleh JVM, sehingga secara otomatis dihancurkan oleh **garbage collector** (bukan oleh programmer).

# Modifier

- Modifier diletakkan pada anatomi kelas, sifatnya optional, digunakan berdasarkan kebutuhan.
- Modifier menunjukkan sifat-sifat tertentu dari : kelasnya, methodnya, atau atributenya.
- Ada beberapa keyword yang digunakan sebagai modifier dan dikelompokkan menjadi :
  - Modifier akses (*public, protected, default, private*)
  - Modifier *final*
  - Modifier *static*
  - Modifier *abstract*

# Modifier akses

- Modifier akses digunakan untuk **membatasi** akses class lain terhadap suatu bagian dari class. (attributes, methods, ataupun class itu sendiri)
- Ada 4 macam modifier akses:
  - **private** → hanya bisa diakses class tersebut
  - **protected** → hanya bisa diakses oleh anak-anaknya
  - **default** → bisa diakses oleh anaknya dan semua yang satu package/satu folder.
  - **public** → umum, bisa dipakai siapa saja.

# Akses Modifier

- Jika tidak disebutkan akses modifiernya berarti default, sifatnya:
  - Dapat diakses pada kelas itu
  - Dapat diakses pada kelas yang sama pada paket (package/directory) yang sama

# Modifier akses

Wilayah Akses	<i>public</i>	<i>protected</i>	<i>default</i>	<i>private</i>
Di kelas yg sama	✓	✓	✓	✓
Beda kelas, di package yg sama	✓	✓	✓	x
Beda kelas, beda package, di kelas turunan	✓	✓	x	x
Beda kelas, beda package, tidak di kelas turunan	✓	x	x	x

# Modifier final

- Untuk membuat sebuah variabel menjadi **konstanta**.
- Kalau modifier final diberikan pada suatu class, maka class tersebut tidak bisa diturunkan/diwariskan.
- Jika diberikan pada suatu method, maka method tersebut tidak bisa diturunkan ke anaknya
- *(Dibahas lebih lanjut pada saat inheritance)*

# Modifier static

- Modifier **static** digunakan untuk membuat sebuah method/attribute bisa diakses *tanpa melakukan instansiasi terlebih dulu*.
- Contoh *System.out.println()* bersifat static artinya untuk memanggil method `println()` tidak harus dilakukan instansiasi dari kelas **System**.
- Nilai attribute yang **static** akan **sama** untuk semua objek.

# Modifier static

```
class StaticNya
{
    static int varStatic;
    int VarNonStatic;

    public static doIt(){
        System.out.println("Ini metode statis");
    }
}
```

# Modifier static

```
class CobaStatic
{
    public static void main(String[] args)
    {
        StaticNya.varStatic = 12;
        System.out.println("Nilai :" + StaticNya.varStatic);

        StaticNya ObjStatic = new StaticNya();
        ObjStatic.varStatic = 3;

        System.out.println("Nilai :" + StaticNya.varStatic);
        StaticNya.doIt();
    }
}
```

# Static

- Kelebihan:
  - Mudah digunakan
  - Tidak perlu instansiasi obyek
  - Dapat digunakan bersama-sama dengan obyek lain
- Kekurangan:
  - Boros memory
  - Tidak bersifat private untuk masing-masing obyek

# Class Design

1. Deklarasikan field (data) sebagai **private** untuk menjaga integritas class. Merupakan prinsip *enkapsulasi*
2. Deklarasikan fungsi dengan **public**, sehingga dapat diakses oleh object lain. Fungsi yang melakukan proses internal yg dideklarasikan dengan private
3. Selalu definisikan **constructor**, lakukan inisialisasi field/data dalam constructor sehingga object dibuat dalam kondisi yang valid
4. Deklarasikan konstanta sebagai **public** bila nilainya akan diakses oleh class lain, bila hanya digunakan untuk internal dideklarasikan **private**
5. Variabel / class / method static boros memory!

# NEXT

- Inheritance