

Design Pattern - Singleton

Singleton in Database Connection and Session Manager JavaFX

Topic List

- Intro to design pattern
 - History
 - Types of design pattern
 - Design Pattern Criticism
- Intro to Singleton Design pattern
 - Initialization strategy of Singleton
 - Different ways to implement singleton method design pattern
 - Make getInstance() static to implement Singleton
 - Make getInstance() synchronized to implement Singleton
 - Eager Instantiation
 - Use “Double Checked Locking”
 - Singleton: pros and cons
- Example

Intro to design pattern

Types of design pattern



Design Pattern- What is it?

- Design pattern (**pola desain**) : adalah solusi masalah yang generic (umum) dalam desain perangkat lunak.
- Seperti **blueprint** siap pakai yang dapat disesuaikan untuk memecahkan masalah desain berulang dalam programming.
- Design pattern != algoritma
 - **Algoritma** : tahapan-tahapan yang strukturnya konkrit, untuk memecahkan masalah tertentu.
 - **Design pattern** : berupa konsep yang abstrak untuk menyelesaikan masalah pada tingkatan arsitektur sistem.

Design Pattern- The history

- Siapa penemunya ?
 - Tidak ada yang tau, karena pola (pattern) muncul karena ia digunakan untuk menyelesaikan masalah yang sering terjadi pada desain OOP.
 - Saat solusi dipergunakan berkali-kali dalam banyak project, maka orang memberikannya nama dan menjelaskannya secara detail.
- Salah satu buku penting (pionir) design pattern berjudul **Design Patterns: Elements of Reusable Object-Oriented Software, 1994** oleh **Erich Gamma, John Vlissides, Ralph Johnson, and Richard Helm (Gang of Four)**.
- Saat ini perkembangannya sangat masif, dan banyak pola bermunculan diluar konsep OOP.

Design Pattern- Why it is important?

- Pola desain adalah **toolkit** yang telah dicoba dan diuji untuk masalah umum dalam desain perangkat lunak.
- Pola desain mendefinisikan “**bahasa umum**” yang dapat digunakan oleh suatu tim untuk berkomunikasi dengan lebih efisien.
- Membuat newbie seperti pro



Design Pattern- criticism

- Design pattern bukanlah **silver bullet**, pola desain yang ada belum tentu cocok dengan project yang sedang dikerjakan.
- Desainer sistem pemula sering menjadikannya dogma, tanpa memahami konteks dan mengadaptasikannya pada project.
- Newbie sering kali memanfaatkan pola desain dimanapun dan kapanpun, membuatnya menjadi ***over engineered***.

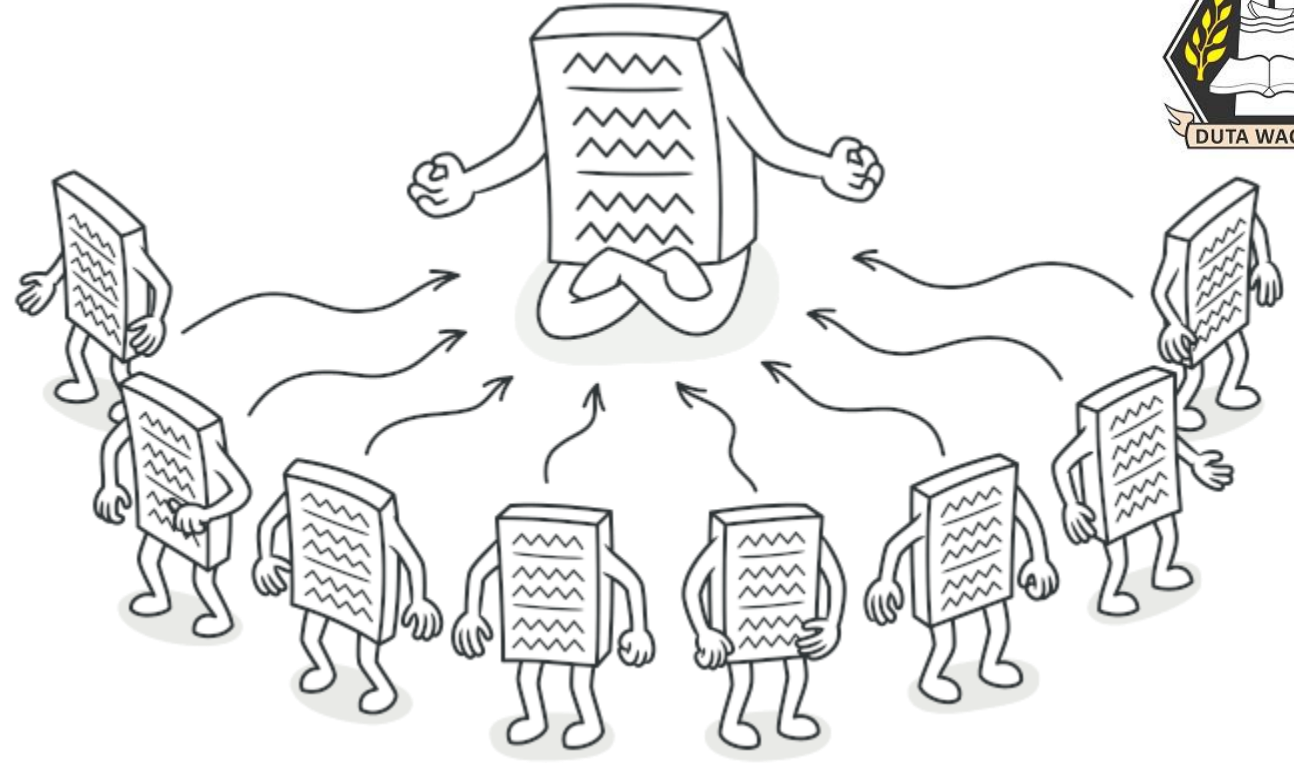
Design Pattern - The catalog

- **Creational patterns** : menyediakan berbagai mekanisme pembuatan objek, yang meningkatkan **fleksibilitas** dan ***code reuseability***.
 - Contoh : **Singleton**, Dependency Injection, Factory Method, Builder, Prototype
- **Structural patterns** : menjelaskan cara merakit objek dan class menjadi **struktur** yang lebih besar sekaligus menjaga struktur tersebut tetap fleksibel dan efisien.
 - Contoh : Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy
- **Behavioral patterns** : Pola-pola ini berkaitan dengan **algoritma** dan penugasan **tanggung jawab** antar objek.
 - Contoh : Observer, Chain of Responsibility, Command, Iterator, Mediator, State, Strategy, Template Method, Visitor.

Singleton design pattern

Initialization Types of Singleton

Different Ways to Implement Singleton Method Design Pattern



Singleton

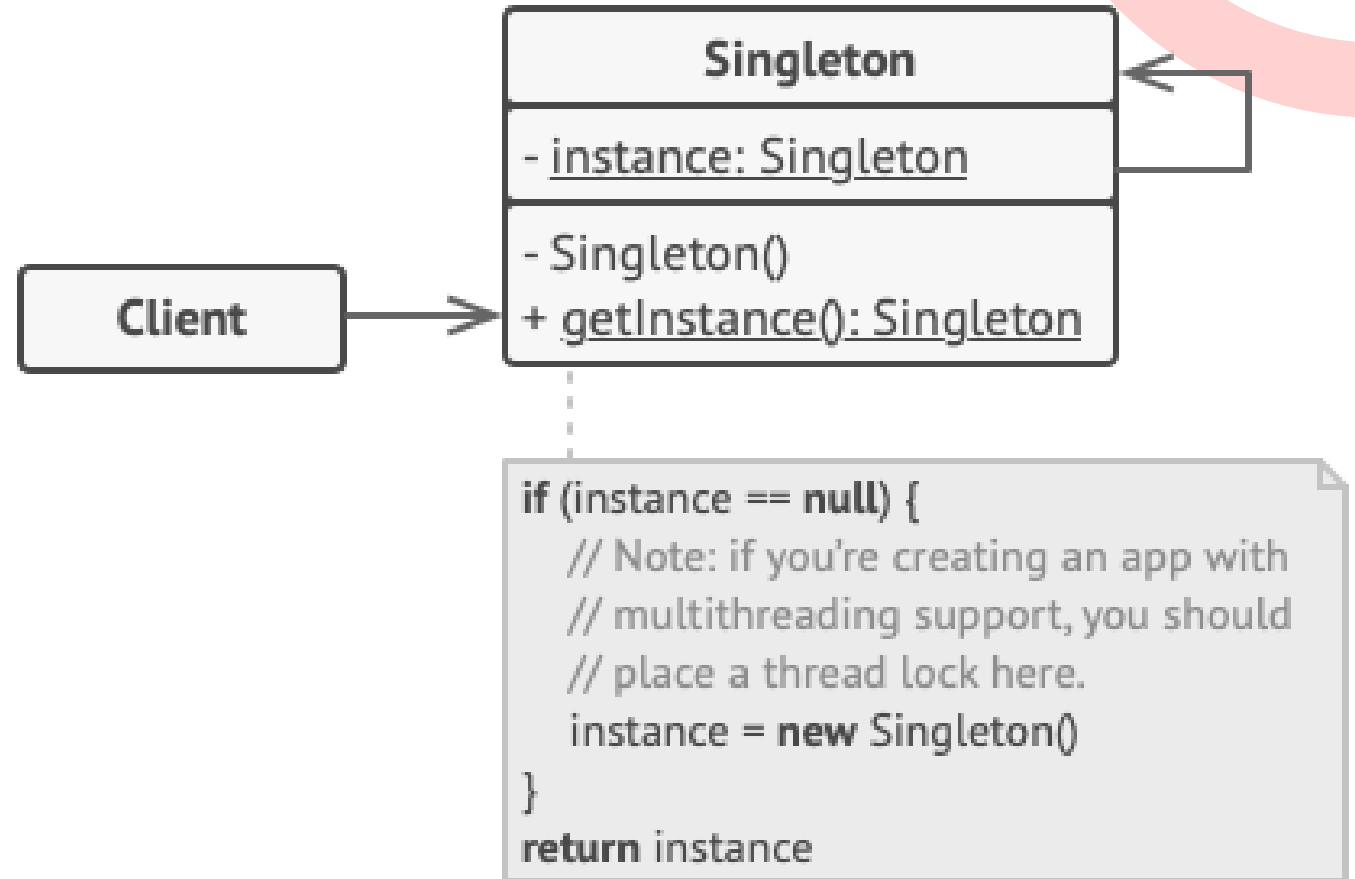
- Pola desain untuk memastikan bahwa suatu kelas hanya memiliki ***satu instance***, sekaligus menyediakan ***titik akses global*** ke instance ini.
- Pola Singleton menyelesaikan dua masalah sekaligus:
 - Pembuatan *instances* yang sama untuk tujuan yang sama (dobel)
 - Mencegah global instances di-overwrite oleh obyek lain

Singleton - implementation

- Semua implementasi Singleton memiliki dua langkah yang sama:
 - Jadikan konstruktor default bersifat **private**, untuk mencegah objek lain menggunakan operator baru dengan kelas Singleton.
 - Buat sebuah **method static** yang bertindak sebagai konstruktor. Method ini memanggil konstruktor private untuk membuat objek dan menyimpannya dalam variable static.
 - NB : kata kunci **static** pada java memungkinkan variable atau method di panggil **tanpa perlu membuat objectnya terlebih dahulu**.
- Sebagai analogi, “sebuah negara hanya boleh memiliki satu pemerintahan yang resmi, sehingga dibutuhkan satu pintu yang resmi pula untuk mengakses semua layanan pemerintah”.

Singleton - implementation

- Kelas Singleton mendeklarasikan method statis **getInstance** yang mengembalikan **instance** yang sama dari kelasnya sendiri.
- Konstruktor Singleton harus disembunyikan dari kode klien. Akss melalui method **getInstance()** menjadi satu-satunya cara untuk mendapatkan objek Singleton.



Jenis bentuk implementasi Singleton

- Terdapat beberapa jenis model implementasi singleton:
 - Method static getInstance() untuk implement Singleton
 - Method synchronized getInstance() untuk implement Singleton
 - Eager Instantiation untuk implement Singleton
 - Double Checked Locking saat implement Singleton

Implementasi - Method static getInstance()

```
// Classical Java implementation of singleton
// design pattern
class Singleton {
    private static Singleton obj;

    // private constructor to force use of
    // getInstance() to create Singleton object
    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj == null)
            obj = new Singleton();
        return obj;
    }
}
```

- Implementasi klasik ini **tidak thread-safe**, karena beberapa thread dapat mengakses singleton diwaktu yang sama dan menciptakan obyek singleton lebih dari satu.

- Deklarasi method **static getInstance()** agar kita dapat memanggilnya tanpa membuat instance kelasnya.
- Pertama kali getInstance() dipanggil, ia membuat objek **tunggal** baru dan setelah itu, ia hanya mengembalikan objek yang **sama**.
- Objek singleton **tidak dibuat** sampai metode getInstance() dipanggil, sehingga disebut **lazy instatiation**.

Implementasi - Method synchronized static getInstance()

```
// Thread Synchronized Java implementation of
// singleton design pattern
class Singleton {
    private static Singleton obj;
    private Singleton() {}

    // Only one thread can execute this at a time
    public static synchronized Singleton getInstance()
    {
        if (obj == null)
            obj = new Singleton();
        return obj;
    }
}
```

- Penggunaan **synchronized** untuk memastikan bahwa hanya **satu thread** pada satu waktu yang dapat mengeksekusi getInstance().
- Kerugian utama dari metode ini adalah penggunaan **synchronized** setiap saat saat membuat objek tunggal membutuhkan **biaya yang mahal** dan dapat menurunkan kinerja program.

- Namun, jika **kinerja** getInstance() tidak penting untuk aplikasi Anda, metode ini memberikan solusi yang aman dan sederhana.

Implementasi - Eager Instantiation

```
// Static initializer based Java implementation of  
// singleton design pattern  
class Singleton {  
    private static Singleton obj = new Singleton();  
    private Singleton() {}  
  
    public static Singleton getInstance() { return obj; }  
}
```

- Instance dari singleton dibuat dalam **inisialisasi statis**.
- JVM mengeksekusi penginisialisasi statis ketika kelas dimuat sehingga metode ini **thread-safe**.
- Gunakan metode ini hanya jika class singleton Anda ringan dan digunakan sepanjang program berjalan.

Implementasi - Double Checked Locking

```
// Double Checked Locking based Java implementation of
// singleton design pattern
class Singleton {
    private static volatile Singleton obj = null;
    private Singleton() {}

    public static Singleton getInstance()
    {
        if (obj == null) {
            // To make thread safe
            synchronized (Singleton.class)
            {
                // check again as multiple threads
                // can reach above step
                if (obj == null)
                    obj = new Singleton();
            }
        }
        return obj;
    }
}
```

- Mendeklarasikan obj dengan kata kunci **volatile** untuk memastikan bahwa beberapa thread mengakses variabel obj dengan benar ketika sedang diinisialisasi ke instance Singleton.
- Metode ini secara drastis mengurangi **overhead** pemanggilan method synchronized.

- Ini adalah metode yang **paling direkomendasikan** karena kita hanya akan mengunci getInstance() sekali saja ketika obj sedang diinsiasi.

Singleton: Pros and Cons

- **Pros:**

- Reduces memory usage: single instance only
- Simplified access and control: a single point of access and synchronization

- **Cons:**

- Reduces scalability (limited) and makes dependency: global / static variable
- Increases complexity and risks: have to synchronize and makes it threads-safe

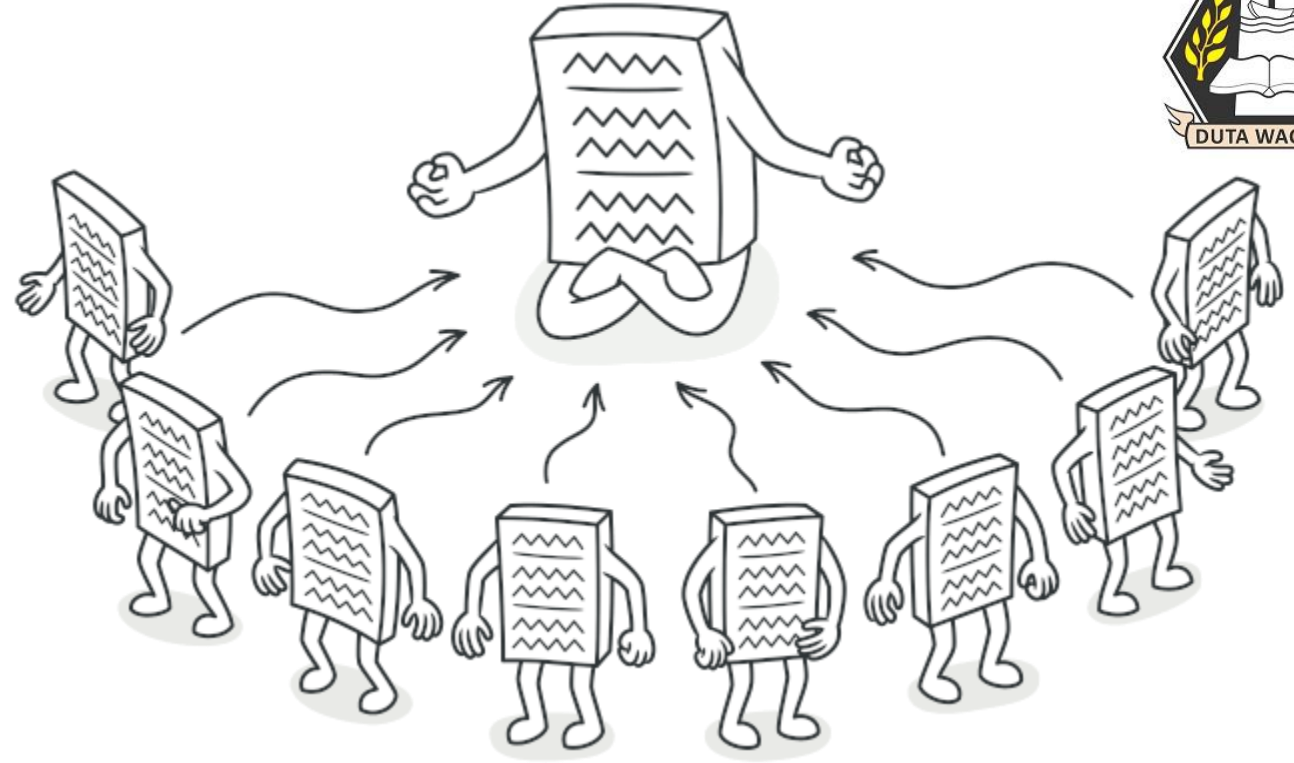
- **Alternative:** Dependency Injection pattern

- next week!

Example

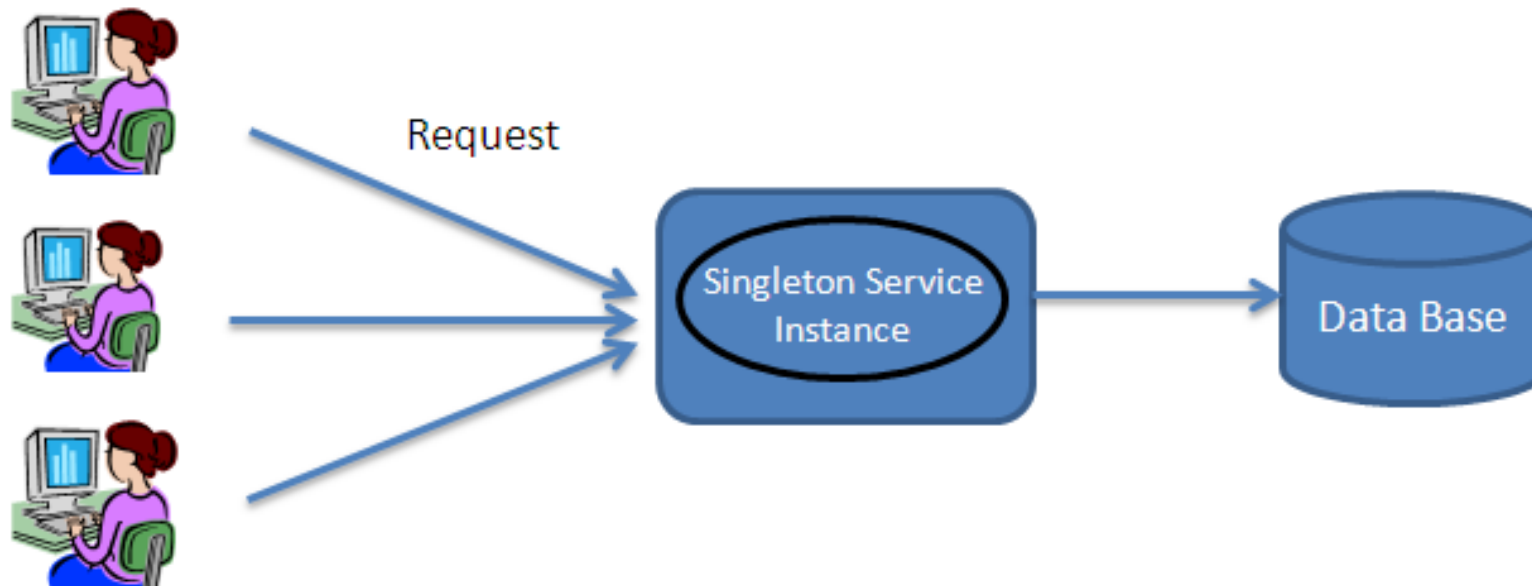
Singleton on DB connection

Singleton on Session management



Example 1 - Singleton on DB connection

- **Singleton** berguna untuk mengatur koneksi ke database karena akan membuat **satu instance koneksi database** bersama yang dapat diakses dari berbagai bagian aplikasi.



Example 1 – Contoh kode

- Mengimplementasikan class **DatabaseUtil** sebagai singleton untuk menyediakan semua koneksi aplikasi ke database.

```
public class DatabaseUtil {  ⚡ Dendy Prtha
    private final String DB_URL = "jdbc:sqlite:nilaimahasiswa.db"; 1 usage
    private Connection connection; 10 usages

    private static volatile DatabaseUtil instance = null; 6 usages

    private DatabaseUtil() { 1 usage  ⚡ Dendy Prtha
    }

    public static DatabaseUtil getInstance() { 6 usages  ⚡ Dendy Prtha
        if (instance == null) {
            // To make thread safe
            synchronized (DatabaseUtil.class) {
                // check again as multiple threads
                // can reach above step
                if (instance == null) {
                    instance = new DatabaseUtil();
                    instance.getConnection();
                    instance.createTable();
                }
            }
        }
        return instance;
    }
}
```

```
public Connection getConnection() { 3 usages  ⚡ Dendy Prtha
    if (connection == null) {
        try {
            connection = DriverManager.getConnection(DB_URL);
        } catch (SQLException e) {
            e.printStackTrace();
            // Handle database connection error
        }
    }
    return connection;
}

public void closeConnection() { no usages  ⚡ Dendy Prtha
    if (connection != null) {
        try {
            connection.close();
        } catch (SQLException e) {
            e.printStackTrace();
            // Handle database connection closure error
        }
    }
}
```

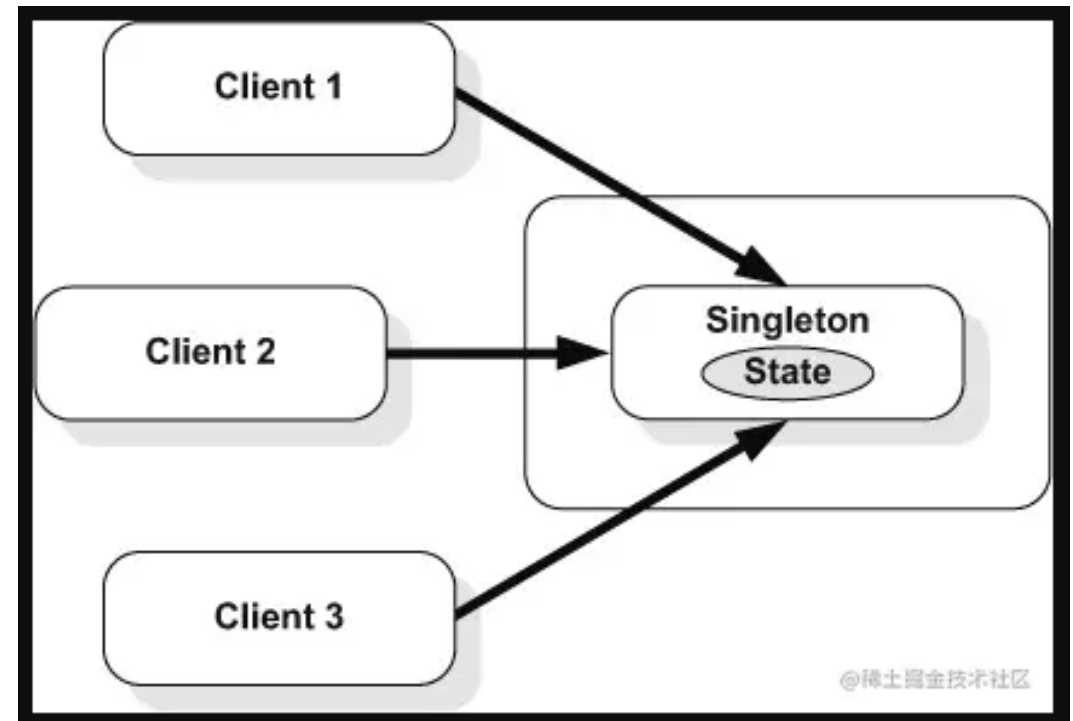
Example 1 – Contoh kode

- Dengan demikian kode menjadi lebih ringkas, dan tidak perlu membuat koneksi setiap akan mengakses database.
- Refactor juga kode pada class lain agar hanya menggunakan singleton class saat mengakses DB

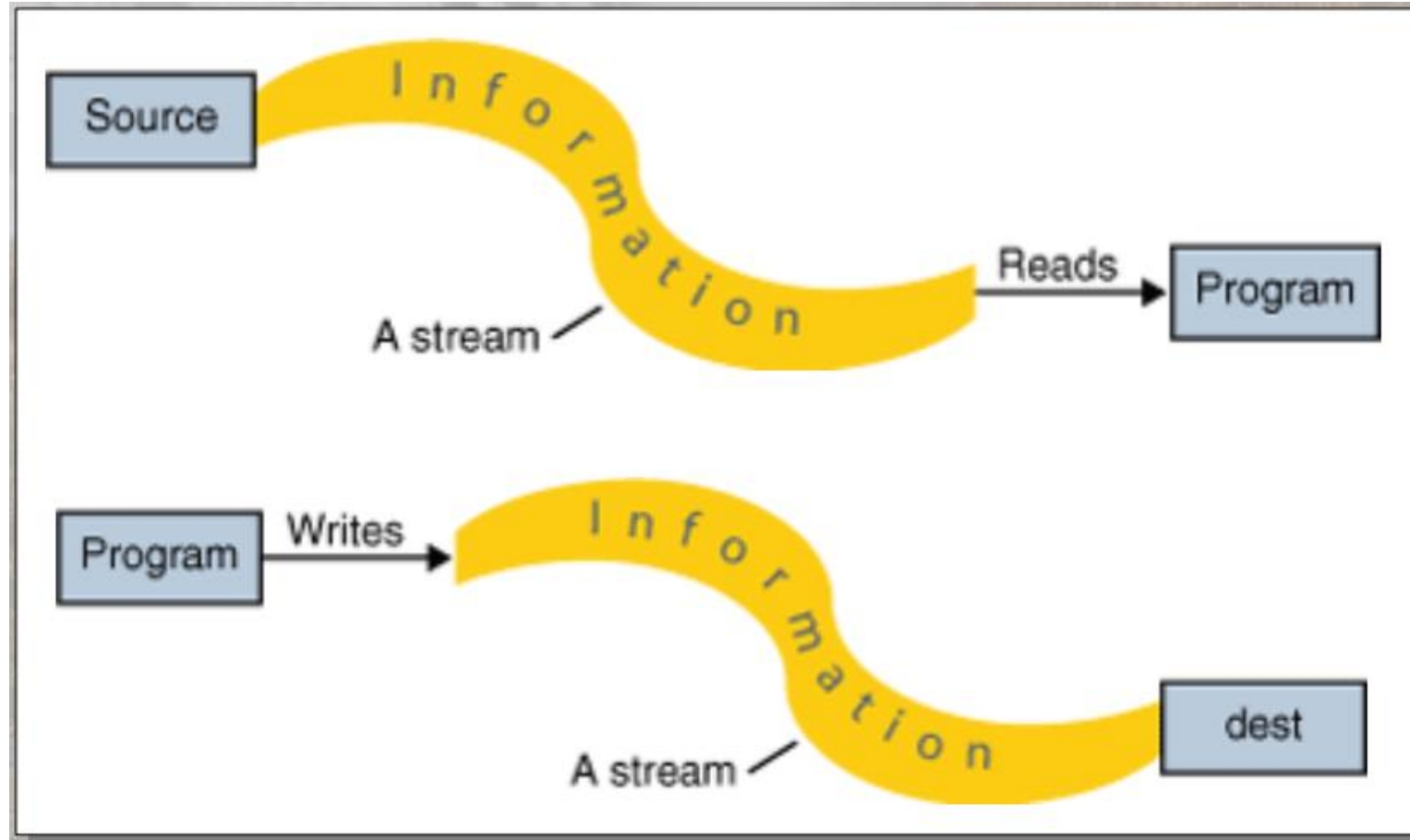
```
public class GrafikMhsController implements Initializable {  * Dendy Prtha *  
    @FXML 1 usage  
    private LineChart lineChart;  
    private Connection connection; 2 usages  
  
    @Override * Dendy Prtha *  
    public void initialize(URL url, ResourceBundle resourceBundle) {  
        connection = DatabaseUtil.getInstance().getConnection();  
        preparedData();  
    }  
  
    private void preparedData() { 1 usage * Dendy Prtha *  
        XYChart.Series series = new XYChart.Series();  
        series.setName("Persebaran Nilai Mahasiswa");  
        String query = "SELECT * FROM mahasiswa";  
        try (PreparedStatement preparedStatement = connection.prepareStatement(query)) {  
            ResultSet resultSet = preparedStatement.executeQuery();  
            while (resultSet.next()) {  
                String nim = resultSet.getString( columnLabel: "nim");  
                String nama = resultSet.getString( columnLabel: "nama");  
                double nilai = resultSet.getDouble( columnLabel: "nilai");  
                byte[] foto = resultSet.getBytes( columnLabel: "foto");  
                Mahasiswa mahasiswa = new Mahasiswa(nim, nama, nilai);  
                mahasiswa.setFoto(foto);  
                series.getData().add(new XYChart.Data(mahasiswa.getNim(), mahasiswa.getNilai()));  
            }  
            lineChart.getData().add(series);  
        } catch (SQLException e) {  
            e.printStackTrace();  
            // Handle database query error  
        }  
    }  
}
```

Example 2 – Session management

- **Session** dibutuhkan untuk mengetahui identitas user yang mengakses sistem dan untuk mengetahui apakah sudah melakukan autentikasi.
- Dengan singleton, kita bisa membuat **single point of truth** untuk menyimpan ID dan status autentikasi pengguna
- Menyimpan file stream session



About Data Stream



Basic Stream Processing

- **Reading**

- open a stream
(defines the source)
- while more information
– read information
- close the stream

- **Provided by:**

- java.io.InputStream
- java.io.Reader

- **Method:**

- read()

- **Writing**

- open a stream
(defines the destination)
- while more information
– write information
- close the stream

- **Provided by:**

- java.io.OutputStream
- java.io.Writer

- **Method:**

- write()
- flush()

Writer: class abstract yang digunakan untuk menulis character-character stream

Reader: class abstract yang digunakan untuk membaca character-character stream

Object Serialization

- *Object Serialization* adalah teknik dimana suatu program dapat menyimpan **status obyek** ke dalam sebuah **file** dan kemudian dapat dipanggil kembali dari file ke memori atau dikirim melalui jaringan.
- **Serialization** menyimpan obyek secara terurut, berbentuk **serialized stream** byte data
- Data byte stream yang telah terurut dapat dibaca kembali di waktu yang akan datang dan kemudian diciptakan kembali menjadi obyek yang disimpan sebelumnya
- Jika sebuah obyek ingin diserialisasi, maka obyek itu harus mengimplementasikan **java.io.Serializable** atau **java.io.Externalizable**
- Untuk menuliskan obyek yang terserialisasi ke file dibutuhkan I/O stream khusus, yaitu menggunakan ***ObjectOutputStream*** yang merupakan subclass dari ***FilterOutputStream***.
- Java serialization **does not** occur for transient or static fields

Example 2 – Contoh kode

- Membuat **session manager**, untuk menyimpan ID dan status autentikasi pengguna.

```
public class SessionManager implements Serializable {  ⚡ Dendy Pr
    private static final long serialVersionUID = 1L;  no usages
    private static final String SESSION_FILE = "session.ser";

    private static volatile SessionManager instance;  5 usages
    private boolean isLoggedIn;  6 usages

    // Private constructor to prevent instantiation from outside
    private SessionManager() {  1 usage  ⚡ Dendy Prtha
        isLoggedIn = false;
    }

    // Static method to get the singleton instance
    public static SessionManager getInstance() {  3 usages  ⚡ Dendy Prtha
        if (instance == null) {
            synchronized (SessionManager.class) {
                if (instance == null) {
                    instance = new SessionManager();
                    instance.createSessionFile();
                }
            }
        }
        return instance;
    }
}
```

```
// Method to check if the session file doesn't exist
public void createSessionFile() {  1 usage  ⚡ Dendy Prtha
    File file = new File(SESSION_FILE);
    if (!file.exists()) {
        saveSession();
    } else {
        loadSession();
    }
}

private void loadSession() {  1 usage  ⚡ Dendy Prtha
    try (ObjectInputStream ois = new ObjectInputStream(new FileInputStream(SESSION_FILE))) {
        SessionManager sessionManager = (SessionManager) ois.readObject();
        this.isLoggedIn = sessionManager.isLoggedIn;
    } catch (IOException | ClassNotFoundException e) {
        System.out.println("Error loading session: " + e.getMessage());
    }
}

private void saveSession() {  3 usages  ⚡ Dendy Prtha
    try (ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream(SESSION_FILE))) {
        oos.writeObject(this);
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Example 2 – Contoh kode


```
// Method to check if user is logged in
public boolean isLoggedIn() { 1 usage  ⚡ De
    return isLoggedIn;
}
```

```
// Method to simulate login
public void login() { 1 usage  ⚡ Dendy Prtha
    isLoggedIn = true;
    saveSession();
}
```

```
// Method to simulate logout
public void logout() { 1 usage  ⚡ Dendy Prtha
    isLoggedIn = false;
    saveSession();
}
```

Example 2 – Contoh kode

- Me-refactor proses inisialisasi aplikasi untuk mengecek apakah user sudah melakukan autentikasi atau belum.

```
public class NilaiMhsApplication extends Application {   Dendry Prtha
    private static Stage primaryStage;  9 usages

    @Override   Dendry Prtha
    public void start(Stage stage) throws IOException {
        primaryStage = stage;
        primaryStage.setTitle("Data Nilai Mahasiswa");
        if (SessionManager.getInstance().isLoggedIn()) {
            primaryStage.setScene(new Scene(loadFXML("form-view")));
        } else {
            primaryStage.setScene(new Scene(loadFXML("login-view")));
        }
        primaryStage.show();
    }
}
```

Example 2 – Contoh kode

- Me-refactor proses autentikasi agar merekam status autentikasi jika login berhasil.

```
public class LoginController {  Dendy Prtha *
    @FXML 1 usage new *
    protected void btnLoginClick() throws IOException {
        Alert alert;
        if (txtUsername.getText().equals(CORRECT_USERNAME) && txtPassword.getText().equals(CORRECT_PASSWORD)) {
            alert = new Alert(Alert.AlertType.INFORMATION);
            alert.setHeaderText("Information");
            alert.setContentText("Login success!!");
            SessionManager.getInstance().login();
            alert.showAndWait();
            NilaiMhsApplication.setRoot( fxml: "form-view", isResizable: false);
        } else {
            alert = new Alert(Alert.AlertType.ERROR);
            alert.setHeaderText("Error");
            alert.setContentText("Login failed!! Please check again.");
            alert.showAndWait();
            txtUsername.requestFocus();
        }
    }
}
```

Example 2 – Contoh kode

- Menambahkan menu logout, untuk deauthenticated user

```
public class NilaiMhsController implements Initializable {  Dendy Prtha *  
    @FXML no usages new *  
    protected void onBtnCloseClick() {  
        // Create a new alert with type Confirmation  
        Alert alert = new Alert(Alert.AlertType.CONFIRMATION);  
        alert.setTitle("Exit & Logout Confirmation");  
        alert.setHeaderText("Are you sure you want to exit?");  
        alert.setContentText("Press OK to exit the application.");  
  
        // Add Yes and No buttons to the alert  
        alert.getButtonTypes().setAll(ButtonType.YES, ButtonType.NO);  
        // Show the alert and wait for user response  
        alert.showAndWait().ifPresent(response -> {  
            if (response == ButtonType.YES) {  
                // User clicked Yes, exit the application  
                SessionManager.getInstance().logout();  
                Platform.exit();  
            }  
        });  
    }  
}
```

Next

- Dependency Injection - Design Patterns