

# Tutorial 6 & 7

---

Praktikum Pemrograman Berbasis Objek  
Asisten IF2210 2023/2024

## Praktikum 5 aman kan?



Semoga bisa tetep aman ya untuk sisa praktikum kedepan :D

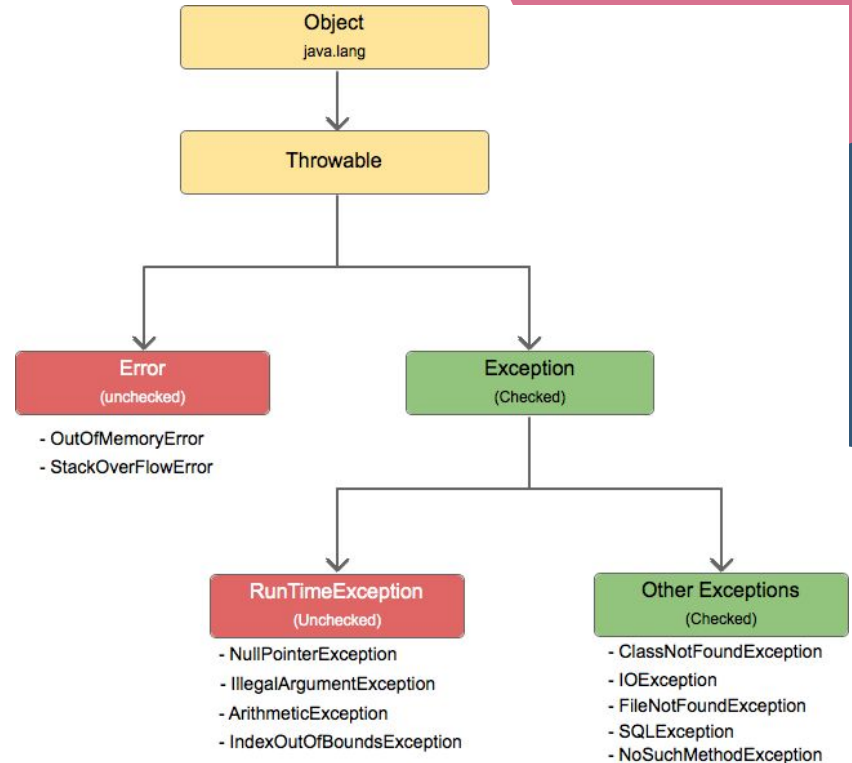
# Kesalahan Umum Praktikum 5

- Salah melihat versi Java API yang digunakan
- Lupa menginisialisasi ArrayList pada constructor
- Melakukan import Deque bawaan dari Java API sehingga terjadi error
- Menginisialisasi exception baru pada `unwrapOrThrow` dan `unwrap`, padahal passed argument dan error yang disimpan sudah bertipe exception
- Kurang teliti meng-handle index dari List
- Tidak banyak menggunakan method yang disediakan class List

# Exception

# Exception

- Mirip dengan C++, Java juga memiliki Exception yang dapat di throw dan catch.
- Namun, object yang di-throw harus merupakan anak dari kelas Throwable. Anak kelas Throwable yang paling mudah digunakan adalah Exception
- Semua anak dari Throwable, kecuali Error dan RuntimeException bersifat checked
- Checked artinya compiler mengharuskan object tersebut perlu di-handle di catch block atau dideklarasikan untuk di-throw pada method signature



# Contoh Kelas Exception

```
class InvalidIndexException extends Exception {  
    public InvalidIndexException(String msg) {  
        super(msg);  
    }  
}
```

# Contoh Penggunaan Exception

```
class Vector {  
    ...  
    public void push(int val) { ... }  
  
    public int get(int idx) throws  
    InvalidIndexException {  
        if (idx < 0 || idx >= this.size) {  
            String msg = String.format("ERROR: The  
index %d you're trying to access is  
inaccessible", idx);  
            throw new InvalidIndexException(msg);  
        }  
        return this.data[idx];  
    }  
}
```

```
class Main {  
    public static void main(String[] args) {  
        Vector v = new Vector(10);  
        v.push(19);  
        v.push(29);  
  
        try {  
            System.out.println(v.get(1));  
            System.out.println(v.get(2));  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

# Catch The Exception!

- Pada bagian try ... catch, perhatikan bahwa kelas yang di-catch adalah kelas Exception.
- Anda tidak harus membuat catch untuk setiap exception yang ada, melainkan bisa langsung melakukan catch terhadap kelas Exception untuk menangkap semua jenis exception yang mungkin muncul. (**ingat kembali materi polymorphism**)



# When To Use Checked vs Unchecked

- Kapan kita menggunakan checked atau unchecked exception?
  - Kita bisa membuat custom unchecked exception dengan inherit dari RuntimeException, sementara checked exception bisa langsung dari class Exception
- The Oracle Java Documentation menyediakan guideline mengenai hal ini
  - “If a client can reasonably be expected to recover from an exception, make it a checked exception. If a client cannot do anything to recover from the exception, make it an unchecked exception.”
- Client dalam hal ini merupakan kelas yang memanggil method yang kita sediakan

# When To Use Checked vs Unchecked

- Contohnya ketika kita membuka file, kita memvalidasi dulu masukan nama file.
  - Jika input file name salah, maka kita bisa throw custom checked exception → yang memanggil method kita bisa aware atas exception ini dan men-catch untuk meminta input user lagi
  - Namun, ketika isi file tidak valid, kita bisa melempar unchecked exception → dengan asumsi input file name benar dan jika file yang akan dibaca tidak valid, maka tidak bisa dilakukan hal lain.

# Design Pattern

# Design Patterns

- Design patterns are typical solutions to common problems in software design.
- Each pattern is like a blueprint that you can customize to solve a particular design problem in your code.
- [refactoring.guru](https://refactoring.guru)

# Design Patterns

## Memandang design pattern (2)

---

- Once you have identified a potentially useful pattern, **try it out**.
- **Read the full text** of the pattern so you get a sense of its **limitations** and **important features**.
- Try to make the pattern fit, but **don't try too hard**.
- Even if the pattern wasn't the right one, you **have not** wasted your time
  - You have **learned something** about the pattern and/or about the problem.
- If a pattern does not fit exactly, **modify it**.
  - Patterns are **suggestions**, not prescriptions.

(Fowler, 1996)

# Design Patterns

## Memandang design pattern (2)

---

- Once you have identified a potentially useful pattern, **try it out**.
- **Read the full text** of the pattern so you get a sense of its **limitations** and **important features**.
- Try to make the pattern fit, but **don't try too hard**.
- Even if the pattern wasn't the right one, you **have not** wasted your time
  - You have **learned something** about the pattern and/or about the problem.
- If a pattern does not fit exactly, **modify it**.
  - Patterns are **suggestions**, not prescriptions.

(Fowler, 1996)






# Kenapa Belajar Design Patterns?

- Design patterns merupakan seperangkat **solusi yang telah banyak dicoba dan diuji** untuk **masalah umum** dalam desain software.
- Namun, terdapat juga kritik terhadap design patterns
  - Unjustified use
    - “If all you have is a hammer, everything looks like a nail.”
    - Ketika pertama belajar design patterns baru, sering kali kita mencoba mengaplikasikannya ke semua hal, meskipun code yang jauh lebih simple pun sudah cukup
  - Inefficient solutions
    - Design patterns didesain sebagai pendekatan yang dapat digunakan pada banyak kasus
    - Seringkali pengimplementasian design patterns itu menjadi dogma → mengaplikasikan design patterns secara strict tanpa mengadaptasikan dalam context masing-masing project

# Jenis-jenis Design Patterns








## Creational patterns

These patterns provide various object creation mechanisms, which increase flexibility and reuse of existing code.

	
Factory Method	Abstract Factory
	
Builder	Prototype
	
Singleton	











## Structural patterns

These patterns explain how to assemble objects and classes into larger structures while keeping these structures flexible and efficient.

	
Adapter	Bridge
	
Composite	Decorator
	
Facade	Flyweight
	
Proxy	

## Behavioral patterns

These patterns are concerned with algorithms and the assignment of responsibilities between objects.

			
Chain of Responsibility	Command	Iterator	Mediator
			
Memento	Observer	State	Strategy
			
Template Method	Visitor		



# Referensi Design Pattern

Beberapa link ini bisa jadi tambahan bacaan sebelum tidur :)

- <https://sourcemaking.com/>
- <https://github.com/kamranahmedse/design-patterns-for-humans>
- <https://refactoring.guru/design-patterns/catalog>

# Multithread

# Multithread

Misalnya kalian lagi buat tugas besar 2 dengan JavaFX:

- Kalian perlu mengupdate waktu setiap detik untuk jam.
- Jika sleep nya kecil maka ada waktu dimana UI akan freeze. Dan update waktu jadi terlalu sering.
- Selain itu, saat membuat laporan, jika data banyak, dan pemrosesan pembuatan pdf lama, UI akan freeze saat memproses pembuatan laporan.
- Nah, untuk menghindari hal tersebut, kalian memerlukan thread lain untuk memproses hal tersebut. Agar main thread/thread khusus UI tidak terganggu dan UI dapat berjalan dengan normal meskipun sedang memproses hal lain.

# Multithread

Itu hanya satu *use case* saja.

Multithread sangat banyak dimanfaatkan di programming, terutama saat satu task bottleneck oleh task lain, padahal keduanya dapat dijalankan bersamaan.

Ibaratnya, kalau kalian nge-carry tugas sendirian itu single thread. Saat kalian bagi tugas itu kalian multi thread. Tentu mengerjakan sendiri dengan bersama kelompok akan ada masalah-masalah baru. Seperti task yang saling berkaitan, saling tunggu-menunggu pekerjaan teman, proses tukar menukar informasi, kontrak kerja antar tugas, pembagian tugas dan sebagainya.

# Multithread

Threading dapat dilakukan dengan 2 cara:

- Membuat kelas yang extends Thread
- Membuat kelas yang implements Runnable

# Extends Thread

```
class FileDownloaderThread extend Thread {  
    // Override  
    public void run() {  
        // Tuliskan fungsi yang akan dijalankan oleh thread  
        // dalam method run  
        downloadFile();  
    }  
}
```

```
...  
Thread fileDownloader = new FileDownloaderThread();  
// Untuk menjalankan thread  
fileDownloader.start();  
...
```

*Pro-Tip:* Program yang akan dijalankan dalam thread lain adalah program yang terdapat pada method **run()** yang dapat di-*override*.

# Implements Runnable

```
class FileDownloader implements Runnable {  
    // Override  
    public void run() {  
        // Tuliskan fungsi yang akan dijalankan oleh thread  
        // dalam method run  
        downloadFile();  
    }  
}
```

```
...  
Runnable fileDownloader = new FileDownloader();  
Thread t1 = new Thread(fileDownloader);  
// Untuk menjalankan thread  
t1.start();  
...
```

# Extends vs Implements

Alasan menggunakan *implement* Runnable:

1. Java tidak memperbolehkan kita *extend* banyak kelas, jadi kita hanya bisa meng-*extend* kelas Thread saja.
2. Dengan *implement* Runnable kita hanya perlu menginstasikannya sekali untuk menjalankannya dalam banyak Thread

```
Runnable runThis = new SimpleRunnable();  
Thread t1 = new Thread(runThis);  
Thread t2 = new Thread(runThis);  
t1.start();  
t2.start();
```

Alasan menggunakan *extend* Thread:

1. *Extend* Thread memungkinkan kita untuk mengubah cara kerja Thread tersebut, seperti menambahkan prosedur baru sebelum Thread di **start()**



# Wait and Notify

# Wait and Notify

Bayangkan kalian diminta membuat sebuah game yang butuh 10 orang untuk dimulai.

Saat invoke start(), Game menunggu 10 orang join game.

Perhatikan kode berikut

```
public class Game {  
    private int playerCount;  
  
    public Game() {  
        this.playerCount = 0;  
    }  
  
    public synchronized void onPlayerJoin() {  
        int prevPlayerCount = this.playerCount;  
        this.playerCount = prevPlayerCount + 1;  
    }  
  
    public synchronized void start() {  
        for (int i = 0; i < 10; i++) {  
            // Misalnya, proses untuk player join butuh waktu lama  
            Thread.sleep(2000);  
            this.onPlayerJoin();  
        }  
        System.out.println("starting game!");  
    }  
}
```

# Wait and Notify

Tapi ini buruk! Karena misal untuk tiap player join butuh 2 detik, maka harus dibutuhkan 20 detik untuk game bisa dimulai!

Karena itu, kita mau memisahkan proses join player menjadi sebuah thread:

# Wait and Notify

```
public class PlayerJoining extends Thread {
    private OnPlayerJoiningListener listener;

    public PlayerJoining(OnPlayerJoiningListener listener) {
        this.listener = listener;
    }

    public void run() {
        // Misalnya, proses untuk player join butuh waktu lama
        Thread.sleep(2000);
        listener.onPlayerJoin();
    }

    public interface OnPlayerJoiningListener {
        void onPlayerJoin();
    }
}
```

# Wait and Notify

```
public class Game implements PlayerJoining.OnPlayerJoiningListener {
    private int playerCount;

    public Game() {
        this.playerCount = 0;
    }

    public void onPlayerJoin() {
        int prevPlayerCount = this.playerCount;
        this.playerCount = prevPlayerCount + 1;
    }

    public void start() {
        while (this.playerCount < 10) {
            // busy waiting, ini akan memakan proses sangat berat
        }
        System.out.println("starting game!");
    }
}
```

# Wait and Notify

Kode tadi jelek, karena ada busy waiting. Resource processor akan termakan hanya untuk loop itu. Masalah ini dapat kita selesaikan dengan *wait* dan *notify*

# Wait and Notify

## *wait()*

Membuat thread yang memanggil fungsi ini menunggu.

## *notify()*

Membuat semua thread yang melakukan *wait()* pada objek yang sama di-resume.

*Pro-tips:* *wait()* dan *notify()* beroperasi pada konteks objek. Jika t1 (Thread) melakukan *wait()* pada objek A dan t2 (thread) melakukan *notify()* pada objek B maka t1 tidak akan di-resume.

# Wait and Notify

```
public class Game implements PlayerJoining.OnPlayerJoiningListener {
    private int playerCount;

    public Game() {
        this.playerCount = 0;
    }

    public void onPlayerJoin() {
        int prevPlayerCount = this.playerCount;
        this.playerCount = prevPlayerCount + 1;
        this.notify();
    }

    public void start() {
        while (this.playerCount < 10) {
            this.wait(); // kode ini blocking, artinya prosesor tidak akan
                        // melanjutkan kecuali ada yang melakukan notify
        }
        System.out.println("starting game!");
    }
}
```



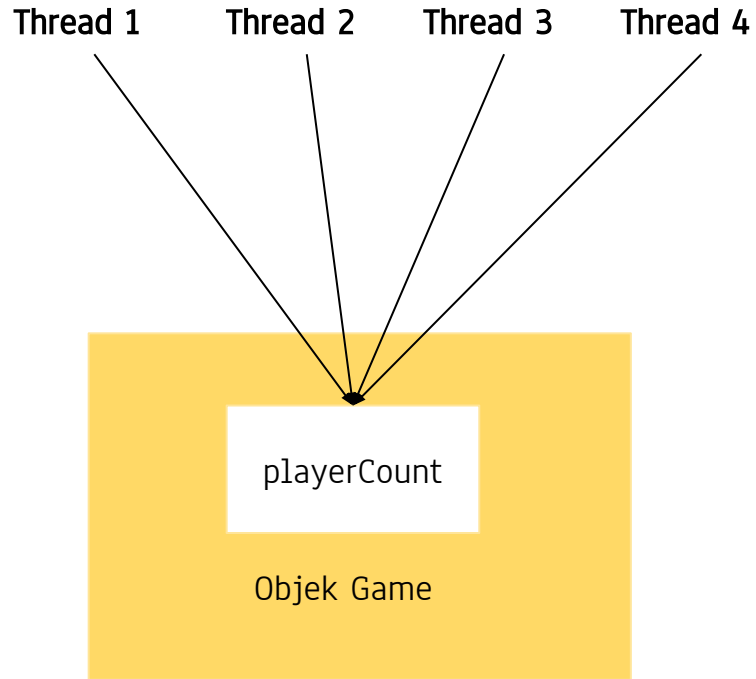
# Wait and Notify

Coba jalankan kode tadi di laptop kalian. Dari awalnya 20 detik, sekarang hanya butuh 2 detik :)

Tapi, perhatikan ada kesalahan pada kode Game:

```
public void onPlayerJoin(String name) {  
    int prevPlayerCount = this.playerCount;  
    this.playerCount = prevPlayerCount + 1;  
    this.notify();  
}
```

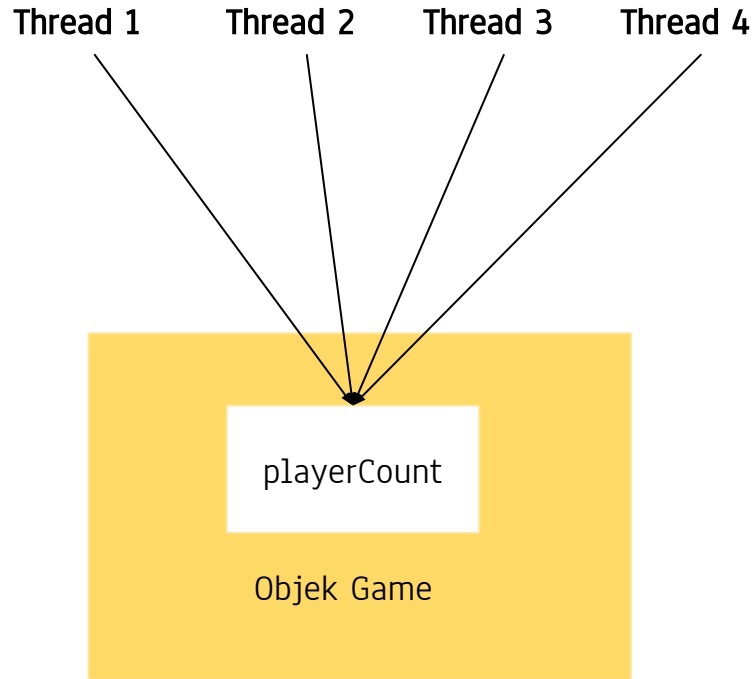
Sudah menemukan kesalahannya?



Thread 1-10 mengakses variabel *counter* yang sama pada objek Runner. Hal ini mereka lakukan ketika memanggil

```
listener.onPlayerJoin()
```

Pada fungsi *onPlayerJoin()* terdapat operasi *increment*. Apa yang terjadi jika operasi *increment* dilakukan dua thread yang berbeda secara bersamaan?



Contoh kasus:

Nilai `playerCount` = 5

Thread 1 membaca nilai `playerCount` // 5

Thread 2 membaca nilai `playerCount` // 5

Thread 1 menambah nilai `playerCount` // 6

Thread 2 menambah nilai `playerCount` // 6

Thread 1 mengassign nilai `playerCount`

Nilai `playerCount` = 6

Thread 2 mengassign nilai `playerCount`

Nilai `playerCount` = 6

Padahal nilai counter seharusnya menjadi 7 ketika dua buah Thread mengakses `onPlayerJoin()`



# Synchronized

# Synchronized

Mengubah kode menjadi

```
public void onPlayerJoin(String name) {  
    this.playerCount++;  
    this.notify();  
}
```

akan menghasilkan masalah yang sama, karena pada dasarnya prosesor akan membreakdown operator increment (++) menjadi 3 step:

1. membaca nilai lama
2. membuat nilai baru yang merupakan nilai lama tambah 1
3. menyimpan nilai baru

# Synchronized

Kasus ini disebut sebagai race condition, ketika 2 thread berebut resource dan menyebabkan perilaku yang tidak diharapkan.

Karena itu, kita bisa memanfaatkan keyword synchronized di java.

Synchronized memastikan agar java tidak akan menjalankan sebuah method dari lebih dari 1 thread sekaligus. Artinya, ketika sebuah method dipanggil oleh sebuah thread, thread lain harus menunggu sampai thread itu selesai menginvoke method itu.

Hasil revisinya ada di slide selanjutnya

# Revisi Wait and Notify

```
public class Game implements PlayerJoining.OnPlayerJoiningListener {
    private int playerCount;

    public Game() {
        this.playerCount = 0;
    }

    public synchronized void onPlayerJoin(String name) {
        int prevPlayerCount = this.playerCount;
        this.playerCount = prevPlayerCount + 1;
        this.notify();
    }

    public synchronized void start() {
        while (this.playerCount < 10) {
            this.wait();
        }
        System.out.println("starting game!");
    }
}
```

# Synchronized

Tips: Jangan meremehkan masalah multi-threading! Race condition, deadlock, dan berbagai macam masalah seperti ini masih sering dijumpai.

Terkadang, kesalahan ada di programmer yang tidak teliti. Misal, bagaimana jika ada 2 thread menambahkan data ke ArrayList secara bersamaan? Kita perlu memeriksa apakah library yang kita gunakan juga *thread safe* (artinya, aman jika diakses oleh beberapa thread sekaligus).



# Timer

# Timer

**Timer** adalah kelas util dari java yang digunakan untuk menjadwalkan aksi secara **periodik** yang akan dijalankan di thread.

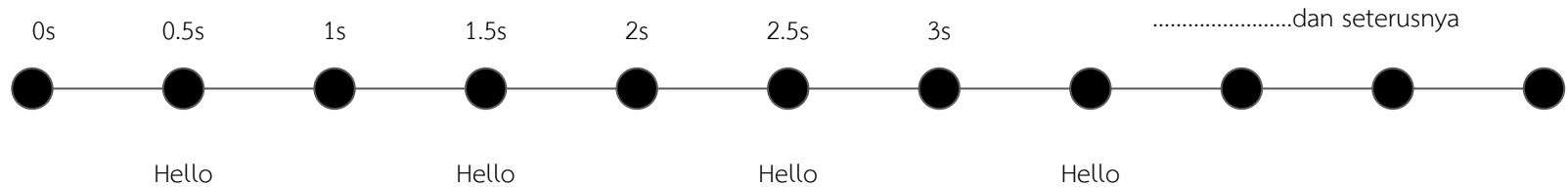
Aksi yang dijadwalkan ini ada didalam kelas **TimerTask**. Dalam kelas ini ada method **run()** yang merupakan aksi yang dijalankan.

Kode berikut akan mencetak “Hello” setiap 1000 ms (1 detik) dan akan jalan dimulai setelah 500ms (0.5 detik)

```
import java.util.Scanner;
import java.util.Timer;
import java.util.TimerTask;

public class Main {
    public static void main(String[] args) {
        Timer timer = new Timer();
        TimerTask timerTask = new TimerTask() {
            public void run() {
                System.out.println("Hello");
            }
        };
        timer.schedule(timerTask, 500, 1000);
    }
}
```

# Timer: Contoh



# Reflection

# Reflection

- Reflection mampu memungkinkan developer mengubah kode saat runtime.
- Contoh yang bisa dilakukan reflection
  - menambahkan method ke sebuah kelas yang ada
  - mengakses member yang private

# Reflection

- Kenapa reflection?
  - Membuat testing
  - Menginjeksi dependency
  - Mengetes kode mahasiswa saat praktikum :)
  - Buat plugin
  - Membuat program magic
  - Mengurangi *nguli*
    - Contoh: di sebuah program Game. Ada 1 package isinya 500 kelas yang extend Skill, dan setiap kelas akan dimasukkan ke Map<String,Skill>, untuk memasukkan ke dalam map tersebut perlu memanggil static field String bernama ID dan instansiasi objek baru. Hal ini bisa dipermudah dengan reflection, sehingga hanya perlu for loop package tersebut dan menggunakan reflection.

# Reflection

- Bayangkan:
  - Tidak perlu membuat Factory untuk tiap kelas yang butuh Factory
  - Meload member kelas dari csv/xml/json/database/lainnya tanpa membuat loader
  - Tidak bisa membuat method / atribut tambahan saat praktikum :)

# Reflection

<https://www.oracle.com/technical-resources/articles/java/javareflection.html>



# Class

- Java memiliki kelas Class (`java.lang.Class`) yang menggambarkan kelas
- Ada beberapa cara mendapatkan kelas:
  - `Class c = Person.class;`
  - `Class c = Class.forName("Person");`
  - `Person p = new Person("Fauzan Rafi Sidiq");`  
`Class c = p.getClass();`
- Setelah mendapatkan kelas, ada banyak yang bisa dilakukan, misal mendapatkan daftar interface, superclass, memanggil constructor, dll
- Daftar lengkapnya ada di <https://docs.oracle.com/javase/8/docs/api/java/lang/Class.html>

# Field

- Java memiliki kelas Field (`java.lang.reflect.Field`) yang menggambarkan field pada sebuah kelas
- Ada beberapa cara mendapatkan method:
  - `Field[] f = c.getFields();`
  - `Field[] f = c.getDeclaredFields();`
  - `Field f = c.getDeclaredField("name");`
- Setelah mendapatkan field, ada banyak yang bisa dilakukan, misal membaca / mengubah nilai, mendapatkan nama field, mendapatkan tipe field, dll
- Daftar lengkapnya ada di <https://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Field.html>

# Method

- Contoh membaca / mengubah field:
  - `Field f = c.getDeclaredField("name");`  
`f.get(obj);`  
`f.set(obj, "Faris-kun");`
    - `obj` merupakan objek kelas yang akan diset namanya
- Untuk mengubah field private, ubah dulu accessnya
  - `f.setAccessible(true);`
- Catatan:
  - `getDeclaredFields` mengembalikan semua private, protected, package, dan public field dari kelas, namun tidak termasuk field yang diinherit
  - `getFields` mengembalikan hanya public field dari kelas, namun termasuk field yang diinherit

# Method

- Sama seperti field, kita bisa menginvoke private method menggunakan
  - `m.setAccessible(true);`
- Contoh menginvoke method:
  - `Method m = c.getDeclaredMethod("getName");`  
`m.invoke(obj);`
  - `Method m = c.getDeclaredMethod("setName", String.class);`  
`m.invoke(obj, "Tuan Mor");`
    - `obj` merupakan objek kelas yang akan diset namanya
- Catatan:
  - `getDeclaredMethods` mengembalikan semua private, protected, package, dan public method dari kelas, namun tidak termasuk method yang diinherit
  - `getMethods` mengembalikan hanya public method dari kelas, namun termasuk method yang diinherit

# Contoh Reflection

- Pernah menggunakan FXML pada JavaFX?
- Fun fact: tidak susah loh membuat fxml.
- Misal ada kelas ini:

```
class CardView {  
    @FXML  
    private Label cardName;  
}
```

- Dan file fxml:

```
<Pane>  
    <children>  
        <Label id="cardName" text="Steve" />  
    </children>  
</Pane>
```

# Contoh Reflection

- Kita bisa membuat Loader yang membaca file fxml.
- Lalu untuk setiap children:
  - Kita buat elemennya (baca semua properti di xml juga)
  - Tambahkan elemennya ke children
  - Cari field di kelas dengan id yang sesuai dan ada anotasi @FXML
  - Set value dari field itu ke elemen yang dibuat
- Jadi FXML di JavaFX tidak “magic”, tapi justru membantu memudahkan kita dalam membuat tampilan GUI

# Catatan

- Banyak hal yang bisa dilakukan dengan reflection
- Tapi untuk praktikum kali ini, kita hanya akan fokus ke Class, Method, dan Field



Sekian.

Ditunggu praktikum dan tutorial selanjutnya.