

---

# Java Object Oriented Programming

Eko Kurniawan Khannedy

# Eko Kurniawan Khannedy

- Technical architect at one of the biggest ecommerce company in Indonesia
- 10+ years experiences
- [www.programmerzamannow.com](http://www.programmerzamannow.com)
- [youtube.com/c/ProgrammerZamanNow](https://youtube.com/c/ProgrammerZamanNow)



---

# Eko Kurniawan Khannedy

- Telegram : [@khannedy](https://t.me/khannedy)
- Facebook : [fb.com/ProgrammerZamanNow](https://www.facebook.com/ProgrammerZamanNow)
- Instagram : [instagram.com/programmerzamannow](https://www.instagram.com/programmerzamannow)
- Youtube : [youtube.com/c/ProgrammerZamanNow](https://www.youtube.com/c/ProgrammerZamanNow)
- Telegram Channel : [t.me/ProgrammerZamanNow](https://t.me/ProgrammerZamanNow)
- Email : echo.khannedy@gmail.com



# Sebelum Belajar

- Java Dasar

---

# Agenda

- Pengenalan OOP
- Object
- Class
- Method
- Pewarisan
- Interface
- Enum
- Exception
- Dan lain-lain

---

# Pengenalan Object Oriented Programming

---

# Apa itu Object Oriented Programming?

- Object Oriented Programming adalah sudut pandang bahasa pemrograman yang berkonsep “objek”
- Ada banyak sudut pandang bahasa pemrograman, namun OOP adalah yang sangat populer saat ini.
- Ada beberapa istilah yang perlu dimengerti dalam OOP, yaitu: Object dan Class

---

# Apa itu Object?

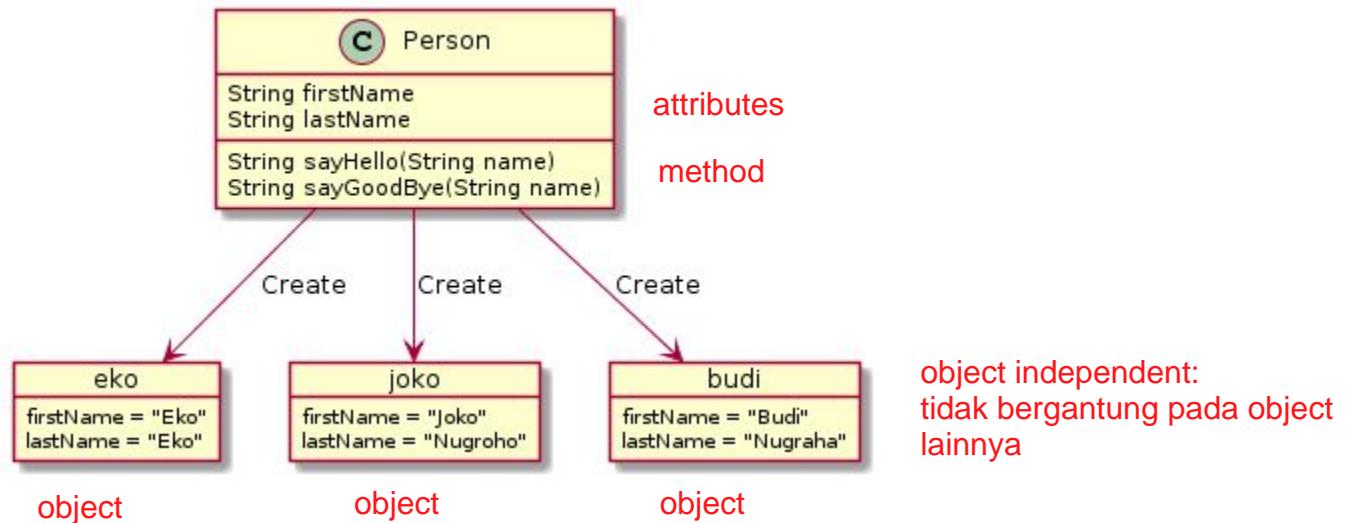
- Object adalah data yang berisi field / properties / **attributes** dan **method** / function / behavior
- Semua data bukan primitif di Java adalah object, dari mulai Integer, Boolean, Character, String dan yang lainnya

---

# Apa itu Class?

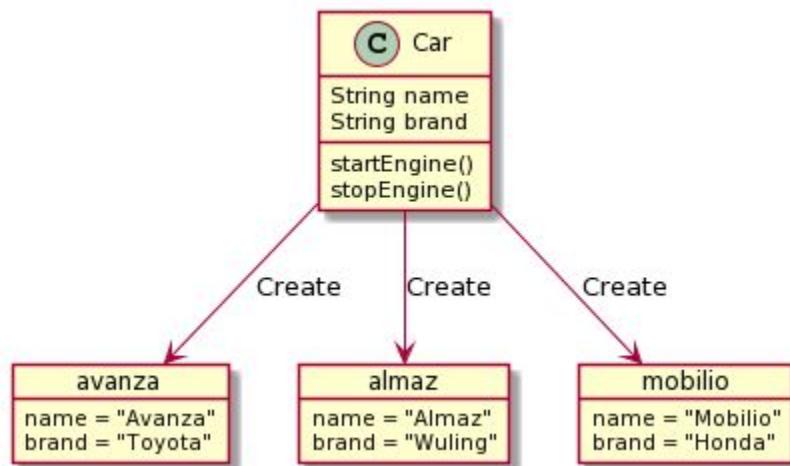
- Class adalah blueprint, prototype atau cetakan untuk membuat Object
- Class berisikan deklarasi semua properties dan functions yang dimiliki oleh Object
- Setiap Object selalu dibuat dari Class
- Dan sebuah Class bisa membuat Object tanpa batas

# Class dan Object : Person



---

# Class dan Object : Car



---

# Class

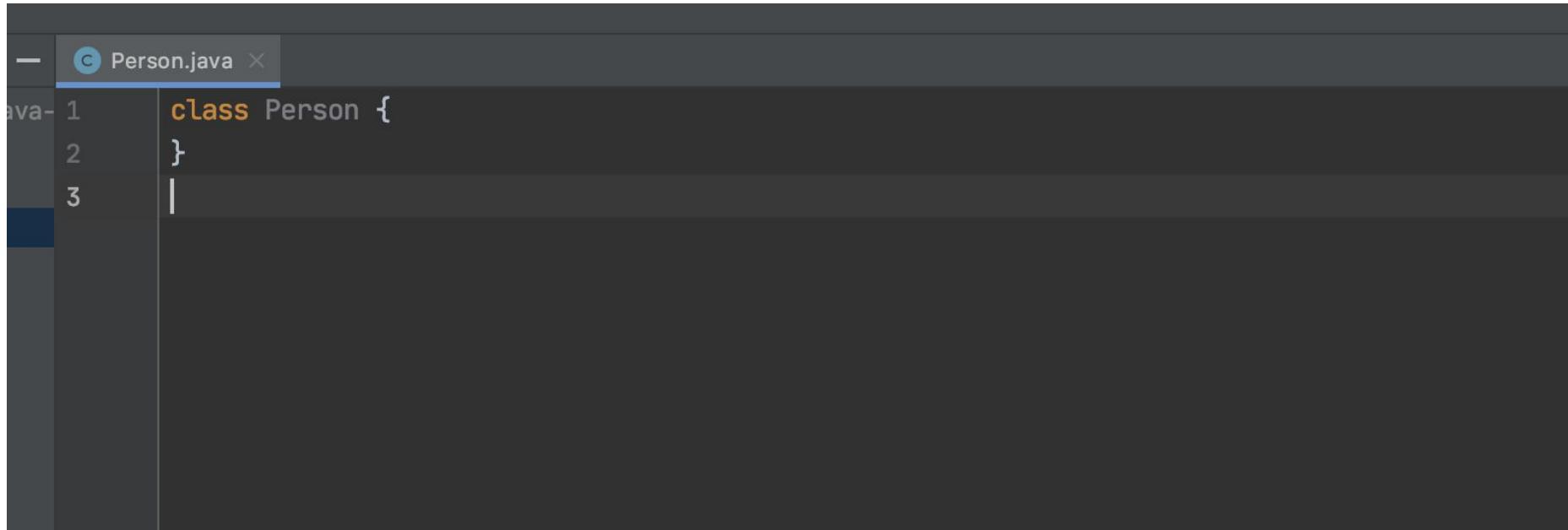
---

# Membuat Class

- Untuk membuat class, kita bisa menggunakan kata kunci class
- Penamaan class biasa menggunakan format CamelCase



# Kode : Class



```
Person.java
1 class Person {
2 }
3 |
```

---

# Object

---

# Membuat Object

- Object adalah hasil instansiasi dari sebuah class
- Untuk membuat object kita bisa menggunakan kata kunci new, dan diikuti dengan nama Class dan kurung ()



# Kode : Object

```
var person1 = new Person();
Person person2 = new Person();

Person person3;
person3 = new Person();

}

}
```

---

# Field

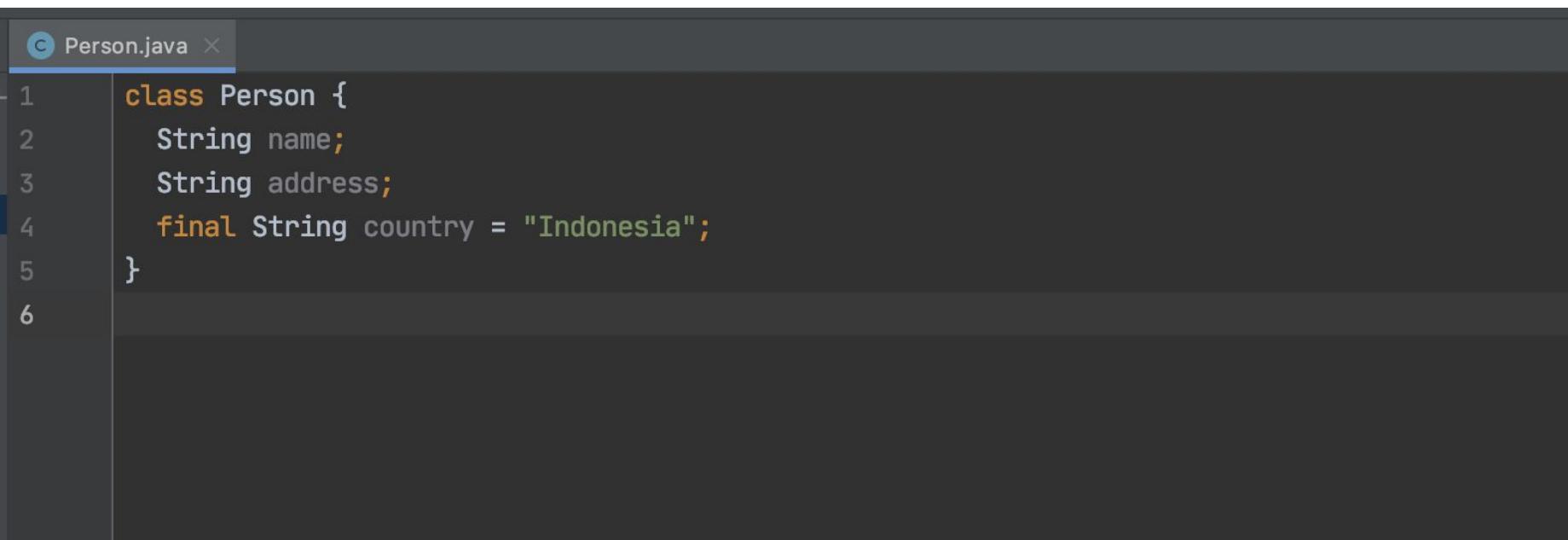
---

# Field

- Fields / Properties / Attributes adalah data yang bisa kita sisipkan di dalam Object
- Namun sebelum kita bisa memasukkan data di fields, kita harus mendeklarasikan data apa aja yang dimiliki object tersebut di dalam deklarasi class-nya
- Membuat field sama seperti membuat variable, namun ditempatkan di block class



## Kode : Field



The screenshot shows a code editor window with a dark theme. The title bar says "Person.java". The code itself is as follows:

```
1 class Person {  
2     String name;  
3     String address;  
4     final String country = "Indonesia";  
5 }  
6
```

---

# Manipulasi Field

- Fields yang ada di object, bisa kita manipulasi. Tergantung final atau bukan.
- Jika final, berarti kita tidak bisa mengubah data field nya, namun jika tidak, kita bisa mengubah field nya
- Untuk memanipulasi data field, sama seperti cara pada variable
- Untuk mengakses field, kita butuh kata kunci . (titik) setelah nama object dan diikuti nama fields nya

---

# Kode : Manipulasi Field

```
var person = new Person();
person.name = "Eko Kurniawan";
person.address = "Subang";
// person.country = "Tidak Bisa Diubah";

System.out.println(person.name);
System.out.println(person.address);
System.out.println(person.country);

}
```

---

# Method

---

# Method

- Selain menambahkan field, kita juga bisa menambahkan method ke object
- Cara dengan mendeklarasikan method tersebut di dalam block class
- Sama seperti method biasanya, kita juga bisa menambahkan return value, parameter dan method overloading di method yang ada di dalam block class
- Untuk mengakses method tersebut, kita bisa menggunakan tanda titik (.) dan diikuti dengan nama method nya. Sama seperti mengakses field

---

## Kode : Method (1)

```
class Person {  
    String name;  
    String address;  
    final String country = "Indonesia";  
  
    void sayHello(String paramName) {  
        System.out.println("Hello " + paramName + ", My Name is " + name);  
    }  
}
```

---

## Kode : Method (2)

```
var person = new Person();
person.name = "Eko Kurniawan";

person.sayHello( paramName: "Budi");

}
```

---

# Constructor

Method yang pertama kali dipanggil saat object dibuat (otomatis gitu lho)

---

# Constructor

- Saat kita membuat Object, maka kita seperti memanggil sebuah method, karena kita menggunakan kurung ()
- Di dalam class Java, kita bisa membuat constructor, constructor adalah method yang akan dipanggil saat pertama kali Object dibuat.
- Mirip seperti di method, kita bisa memberi parameter pada constructor
- Nama constructor harus sama dengan nama class, dan tidak membutuhkan kata kunci void atau return value

---

# Kode : Membuat Constructor

```
1  class Person {  
2      String name;  
3      String address;  
4      final String country = "Indonesia";  
5  
6      Person(String paramName, String paramAddress) {  
7          name = paramName;  
8          address = paramAddress;  
9      }  
10  
11     void sayHello(String paramName) {  
12         System.out.println("Hello " + paramName + ", My Name is " + name);  
13     }  
14 }
```

---

## Kode : Menggunakan Constructor

```
3  
4  
5     var person = new Person( paramName: "Eko", paramAddress: "Subang");  
6     person.name = "Eko Kurniawan";  
7  
8     person.sayHello( paramName: "Budi");  
9  
0 }  
1 }
```

---

# Constructor Overloading

---

# Constructor Overloading

- Sama seperti di method, di constructor pun kita bisa melakukan overloading
- Kita bisa membuat constructor lebih dari satu, dengan syarat tipe data parameter harus berbeda, atau jumlah parameter harus berbeda



# Kode : Constructor Overloading

```
6  Person(String paramName, String paramAddress) {  
7      name = paramName;  
8      address = paramAddress;  
9  }  
10  
11 Person(String paramName) {  
12     name = paramName;  
13 }  
14  
15 Person() {  
16 }  
17 }
```

---

## Kode : Menggunakan Constructor

```
5     var person1 = new Person( paramName: "Eko", paramAddress: "Subang");
6     var person2 = new Person( paramName: "Eko");
7     var person3 = new Person();|
```

```
9   }
0 }
```

---

# Memanggil Constructor Lain

- Constructor bisa memanggil constructor lain
- Hal ini memudahkan saat kita butuh menginisialisasi data dengan berbagai kemungkinan
- Cara untuk memanggil constructor lain, kita hanya perlu memanggilnya seperti memanggil method, namun dengan kata kunci this



# Kode : Memanggil Constructor Lain

```
Person(String paramName, String paramAddress) {  
    name = paramName;  
    address = paramAddress;  
}  
  
Person(String paramName) {  
    this(paramName, paramAddress: null);  
}  
  
Person() {  
    this(paramName: null);  
}
```

---

# Variable Shadowing

SEDIKIT BINGUNG HM...

---

# Variable Shadowing

- Variable shadowing adalah kejadian ketika kita membuat nama variable dengan nama yang sama di scope yang menutupi variable dengan nama yang sama di scope diatasnya
- Ini biasa terjadi seperti kita membuat nama parameter di method sama dengan nama field di class
- Saat terjadi variable shadowing, maka secara otomatis variable di scope diatasnya tidak bisa diakses

# Kode : Variable Shadowing

```
final String country = "Indonesia",  
  
Person(String name, String address) {  
    name = name; // field nama tidak berubah  
    address = address; // field address tidak berubah  
}  
  
void sayHello(String name) {  
    System.out.println("Hello " + name + ", My Name is " + name); // field name tidak diakses  
}  
  
void sayHello() {  
    this.sayHello(name: "Bos");  
}
```

---

# This Keyword

this digunakan untuk mengakses object saat ini  
jika dalam 1 class ada object -> String name  
selain bisa dipanggil dengan -> name  
bisa juga dipanggil menggunakan -> this.name

---

# This Keyword

- Saat kita membuat kode di dalam block constructor atau method di dalam class, kita bisa menggunakan kata kunci this untuk mengakses object saat ini
- Misal kadang kita butuh mengakses sebuah field yang namanya sama dengan parameter method, hal ini tidak bisa dilakukan jika langsung menyebut nama field, kita bisa mengakses nama field tersebut dengan kata kunci this
- This juga tidak hanya digunakan untuk mengakses field milik object saat ini, namun juga bisa digunakan untuk mengakses method
- This bisa digunakan untuk mengatasi masalah variable shadowing

# Kode : This Keyword

```
5
6     Person(String name, String address) {
7         this.name = name;
8         this.address = address;
9     }
10
11    void sayHello() {
12        this.sayHello( name: "Bos");
13    }
14
15    void sayHello(String name) {
16        System.out.println("Hello " + name + ", My Name is " + this.name);
17    }
```

---

# Inheritance

---

# Inheritance

- Inheritance atau pewarisan adalah kemampuan untuk menurunkan sebuah class ke class lain
- Dalam artian, kita bisa membuat class Parent dan class Child
- Class Child, hanya bisa punya satu class Parent, namun satu class Parent bisa punya banyak class Child
- Saat sebuah class diturunkan, maka semua field dan method yang ada di class Parent, secara otomatis akan dimiliki oleh class Child
- Untuk melakukan pewarisan, di class child, kita harus menggunakan kata kunci extends lalu diikuti dengan nama class parent nya.

# Kode : Inheritance

```
1 class Manager {  
2     String name;  
3  
4     void sayHello(String name) {  
5         System.out.println("Hello " + name + ", My Name is " + this.name);  
6     }  
7 }  
8  
9 class VicePresident extends Manager {  
10 }  
11 }
```



---

# Kode : Mengakses Method Parent

```
4
5     var manager = new Manager();
6     manager.name = "Eko";
7     manager.sayHello( name: "Budi");
8
9     var vicePresident = new VicePresident();
10    vicePresident.name = "Kurniawan";
11    vicePresident.sayHello( name: "Budi");
12
13 }
14 }
```

---

# Method Overriding

---

# Method Overriding

- Method overriding adalah kemampuan mendeklarasikan ulang method di child class, yang sudah ada di parent class
- Saat kita melakukan proses overriding tersebut, secara otomatis ketika kita membuat object dari class child, method yang di class parent tidak bisa diakses lagi

Berbeda dengan overloading

>> overloading -> membuat method yg sama dalam 1 class & semua nya bisa terus digunakan

>> overriding -> membuat method yang dalam dalam child dari parent, yang parent jadi tidak bisa digunakan



# Kode : Method Overriding

```
1 class Manager {  
2     String name;  
3  
4     void sayHello(String name) {  
5         System.out.println("Hello " + name + ", My Name is Manager " + this.name);  
6     }  
7 }  
8  
9 class VicePresident extends Manager {  
10  
11    void sayHello(String name) {  
12        System.out.println("Hello " + name + ", My Name is VP " + this.name);  
13    }  
14 }
```

---

# Kode : Mengakses Method Overriding

```
var manager = new Manager();
manager.name = "Eko";
manager.sayHello( name: "Budi");

var vicePresident = new VicePresident();
vicePresident.name = "Kurniawan";
vicePresident.sayHello( name: "Budi");

}

}
```

---

# Super Keyword

---

# Super Keyword

- Kadang kita ingin mengakses method yang terdapat di class parent yang sudah **terlanjur kita override di class child**
- Untuk mengakses method milik class parent, kita bisa menggunakan kata kunci super
- Sederhananya, **super digunakan untuk mengakses class parent**
- Tidak hanya method, field milik parent class pun bisa kita akses menggunakan kata kunci super

mirip seperti this

>> jika this untuk mengakses object dalam 1 class

>> jika super untuk mengakses object di child dari parent

# Kode : Super Keyword

```
1  class Shape {  
2      int getCorner(){  
3          return 0;  
4      }  
5  }  
16  
17  class Rectangle extends Shape {  
18      int getCorner() {  
19          return 4;  
20      }  
21  
22      int getParentCorner(){  
23          return super.getCorner();  
24      }  
25  }
```

---

# Kode : Mengakses Super Keyword

```
var rectable = new Rectangle();

System.out.println(rectable.getCorner());
System.out.println(rectable.getParentCorner());

}
```

---

# Super Constructor

---

# Super Constructor

- Tidak hanya untuk mengakses method atau field yang ada di parent class, kata kunci **super** juga **bisa digunakan untuk mengakses constructor**
- Namun syaratnya untuk mengakses parent class constructor, kita harus mengaksesnya di dalam class child constructor
- Jika sebuah class parent tidak memiliki constructor yang tidak ada parameter-nya (tidak memiliki default constructor), maka class child wajib mengakses constructor class parent tersebut.  
**walaupun mengakses dengan mengisi null pada parameternya juga ndapapa oce**

parent memiliki constructor yang tidak ada parameter nya (default constructor) >> ini tidak wajib mengakses parent constructor nya (kebalikan poin 3 biar lebih mudah dipahami oleh saya HUEHEHE

---

# Kode : Super Constructor

```
Manager(String name){  
    this.name = name;  
}  
}  
  
class VicePresident extends Manager {  
  
    VicePresident(String name){  
        super(name);  
    }  
}
```

---

# Kode : Menggunakan Super Constructor

```
var manager = new Manager( name: "Eko");
manager.sayHello( name: "Budi");

var vicePresident = new VicePresident( name: "Kurniawan");
vicePresident.sayHello( name: "Budi");

}
```

---

# Object Class

---

# Object Class

- Di Java, setiap class yang kita buat secara otomatis adalah turunan dari class Object
- Walaupun tidak secara langsung kita eksplisit menyebutkan extends Object, tapi secara otomatis Java akan membuat class kita extends Object
- Bisa dikatakan class Object adalah superclass untuk semua class yang ada di Java

# Isi Class Object

The screenshot shows the Java Object class structure and its implementation. On the left, the 'Structure' view displays the class hierarchy and member list. The 'Object' class has the following members:

- m `Object()`
- m `getClass(): Class<?>`
- m `hashCode(): int`
- m `equals(Object): boolean`
- m `clone(): Object`
- m `toString(): String`
- m `notify(): void`
- m `notifyAll(): void`
- m `wait(): void`
- m `wait(long): void`
- m `wait(long, int): void`
- m `finalize(): void`

On the right, the code implementation is shown:

```
35 * @author unasccribed
36 * @see java.lang.Class
37 * @since 1.0
38 */
39 public class Object {
40
41 /**
42 * Constructs a new object.
43 */
44 @HotSpotIntrinsicCandidate
45 public Object() {}
46 }
```

---

## Kode : Menggunakan Class Object

```
var manager = new Manager( name: "Eko");
System.out.println(manager); // otomatis memanggil method toString()
System.out.println(manager.toString());
```

```
}
```

---

# Polymorphism

---

# Polymorphism

- Polymorphism berasal dari bahasa Yunani yang berarti banyak bentuk.
- Dalam OOP, Polymorphism adalah kemampuan sebuah object berubah bentuk menjadi bentuk lain
- Polymorphism erat hubungannya dengan Inheritance

# Kode : Inheritance

```
1  class Employee {  
2      String name;  
3      Employee(String name){  
4          this.name = name;  
5      }  
6  }  
  
7  
8  class Manager extends Employee{  
9      Manager(String name){  
10         super(name);  
11     }  
12 }
```

```
13  
14  class VicePresident extends Manager {  
15      VicePresident(String name){  
16          super(name);  
17      }  
18  }  
19  
20
```



# Kode : Polymorphism

```
Employee employee = new Employee( name: "Eko");
employee.sayHello( name: "Budi");

employee = new Manager( name: "Eko");
employee.sayHello( name: "Budi");

employee = new VicePresident( name: "Eko");
employee.sayHello( name: "Budi");

}
```

# Kode : Method Polymorphism

```
sayHello(new Employee( name: "Eko"));
sayHello(new Manager( name: "Eko"));
sayHello(new VicePresident( name: "Eko"));

}

@ static void sayHello(Employee employee) {
    System.out.println("Hello " + employee.name);
}
```

---

# Type Check & Casts

---

# Type Check & Casts

- Sebelumnya kita sudah tau cara melakukan konversi tipe data (casts) di tipe data primitif
- Casts juga bisa digunakan untuk tipe data bukan primitif
- Namun agar aman, sebelum melakukan casts, pastikan kita melakukan type check (pengecekan tipe data), dengan menggunakan kata kunci instanceof
- Hasil operator instanceof adalah boolean, true jika tipe data sesuai, false jika tidak sesuai

---

# Kode : Type Check & Casts

```
static void sayHello(Employee employee) {  
    if (employee instanceof VicePresident) {  
        VicePresident vicePresident = (VicePresident) employee;  
        System.out.println("Hello VP " + vicePresident.name);  
    } else if (employee instanceof Manager) {  
        Manager manager = (Manager) employee;  
        System.out.println("Hello Manager " + manager.name);  
    } else {  
        System.out.println("Hello VP " + employee.name);  
    }  
}
```

---

# Variable Hiding

---

# Variable Hiding

- Variable hiding merupakan masalah yang terjadi ketika kita membuat nama field sama di class child dengan nama field di class parent
- Tidak ada yang namanya field overriding, ketika kita buat ulang nama field di class class, itu berarti variable hiding
- Untuk mengatasi variable hiding, caranya kita bisa menggunakan super keyword
- Yang membedakan variable hiding dan method overriding adalah ketika sebuah object di casts
- Saat object di casts, method akan tetap mengakses method overriding, namun variable akan mengakses variable yang ada di class nya

---

# Kode : Variable Hiding

```
① class Parent {  
    String name;  
    void doIt(){  
        System.out.println("Do it from parent");  
    }  
}  
  
class Child extends Parent {  
    String name;  
    void doIt(){  
        System.out.println("Do it from child");  
    }  
}
```

---

# Kode : Variable Hiding vs Method Overriding

```
Child child = new Child();
child.name = "Eko";
child.doIt();
System.out.println(child.name);

Parent parent = (Parent) child;
parent.doIt();
System.out.println(parent.name);

}
```

---

# Package

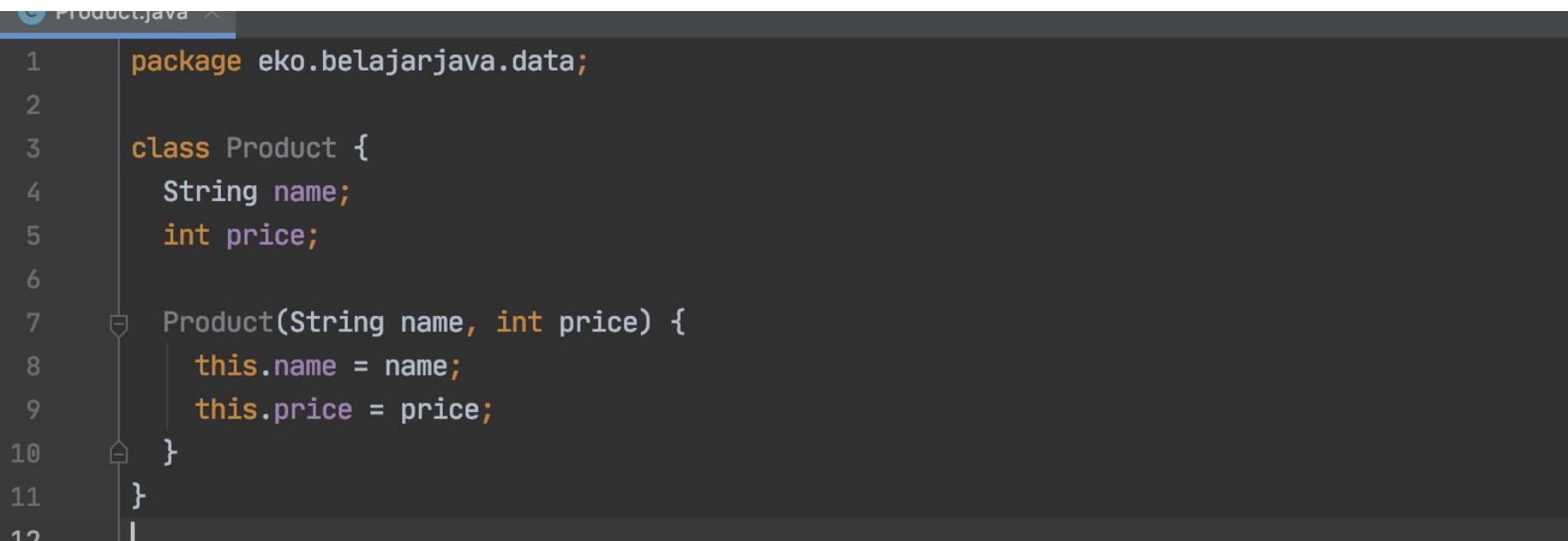
---

# Package

- Saat kita membuat aplikasi, bisa dipastikan kita akan banyak sekali membuat class
- Jika class terlalu banyak, kadang akan menyulitkan kita untuk mencari atau mengklasifikasikan jenis-jenis class
- Java memiliki fitur package, yaitu fitur mirip folder / direktori, dimana kita bisa menyimpan class-class kita di dalam package
- Sama seperti folder / direktori, package juga bisa nested, kita bisa menggunakan tanda titik (.) untuk membuat nested package
- Ketika kita menyimpan class di dalam package, maka diatas file Java nya, kita wajib menyebutkan nama package nya

---

# Kode : Package



A screenshot of a Java code editor showing a file named `Product.java`. The code defines a class `Product` with a constructor that takes `String name` and `int price`, and initializes them. The code is color-coded, with `package`, `class`, and `this` keywords in orange, and variable names in purple.

```
1 package eko.belajarjava.data;
2
3 class Product {
4     String name;
5     int price;
6
7     Product(String name, int price) {
8         this.name = name;
9         this.price = price;
10    }
11 }
12
```

---

# Access Modifiers

---

# Access Modifier

- Access modifier adalah kemampuan membuat class, field, method dan constructor dapat diakses dari mana saja
- Sebelumnya teman-teman sudah melihat 2 access modifier, yaitu public dan default (no-modifier)
- Sekarang kita akan bahas access modifier lainnya



# Access Level

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
<i>no modifier</i>	Y	Y	N	N
private	Y	N	N	N

---

# Public Class

- Saat kita membuat public class, kita hanya bisa membuat 1 public class di 1 file java
- Selain itu, nama public class harus sama dengan nama file

---

# Kode : Access Modifier (1)

```
1 package eko.belajarjava.data;  
2  
3 public class Product {  
4     protected String name;  
5     protected int price;  
6  
7     public String getName() {  
8         return name;  
9     }  
10  
11    public int getPrice() {  
12        return price;
```

---

## Kode : Access Modifier (2)

```
1 package eko.belajarjava.data;  
2  
3 ► public class ProductApp {  
4 ►   □ public static void main(String[] args) {  
5     Product product = new Product();  
6     product.name = "Indomie"; // bisa diakses karena di package sama  
7     product.price = 2000; // // bisa diakses karena di package sama  
8  
9     System.out.println(product.getName());  
10    System.out.println(product.getPrice());  
11  }  
12 }
```

---

# Import

---

# Import

- Import adalah kemampuan untuk menggunakan class yang berada di package yang berbeda
- Syarat class yang bisa digunakan jika package nya berbeda adalah class yang harus public



# Kode : Import

```
1 package eko.belajarjava.web;  
2  
3 import eko.belajarjava.data.Product;  
4  
5 ► public class ProductWeb {  
6   ►   public static void main(String[] args) {  
7     Product product = new Product();  
8     product.name = "Indomie"; // error, beda package  
9     product.price = 2000; // // error, beda package  
10  
11    System.out.println(product.getName());  
12    System.out.println(product.getPrice());
```

---

# Import Semua Package

- Jika kita ingin mengimport semua class di dalam sebuah package, kita bisa menggunakan tanda \*, misal
- import eko.belajar.oop.data.\*

---

# Default Import

- Secara default, semua class yang ada di package java.lang sudah auto import, jadi kita tidak perlu melakukan import secara manual
- Contoh class String, Integer, Long, Boolean, dan lain-lain terdapat di package java.lang. Oleh karena itu, kita tidak perlu meng-import nya secara manual

---

# Abstract Class

---

# Abstract Class

- Saat kita membuat class, kita bisa menjadikan sebuah class sebagai abstract class.
- Abstract class artinya, class tersebut tidak bisa dibuat sebagai object secara langsung, hanya bisa diturunkan
- Untuk membuat sebuah class menjadi abstract, kita bisa menggunakan kata kunci abstract sebelum kata kunci class
- Dengan demikian abstract class bisa kita gunakan sebagai kontrak untuk class child

# Kode : Abstract Class

```
Location.java
1 package eko.belajarjava.data;
2
3 public abstract class Location {
4     String name;
5 }
6
```

```
City.java
1 package eko.belajarjava.data;
2
3 public class City extends Location {
4 }
```

---

# Kode : Membuat Abstract Class

```
2
3 import eko.belajarjava.data.City;
4 import eko.belajarjava.data.Location;
5
6 ► public class AbstractClassApp {
7 ►   public static void main(String[] args) {
8
9     var location = new Location(); // error
10    var city = new City();
11
12  }
13
14 }
```

---

# Abstract Method

---

# Abstract Method

- Saat kita membuat class yang abstract, kita bisa membuat abstract method juga di dalam class abstract tersebut
- Saat kita membuat sebuah abstract method, kita tidak boleh membuat block method untuk method tersebut
- Artinya, abstract method wajib di override di class child
- Abstract method tidak boleh memiliki access modifier private

# Kode : Abstract Method

```
3 ⚡ public abstract class Animal {  
4     public String name;  
5 ⚡     public abstract void run();  
6 }
```

Cat.java ×

```
1 package eko.belajarjava.data;  
2  
3 public class Cat extends Animal {  
4 ⚡     ↗     public void run() {  
5         System.out.println("Cat " + name + " is running");  
6     }
```

---

# Kode : Menggunakan Abstract Method

```
5  
6 ►  public class AbstractMethodApp {  
7 ►    □  public static void main(String[] args) {  
8  
9      Animal animal = new Cat();  
10     animal.name = "Eko";  
11     animal.run();  
12    }  
13  }  
14 }  
15 |
```

---

# Getter dan Setter

---

# Encapsulation

- Encapsulation artinya memastikan data sensitif sebuah object tersembunyi dari akses luar
- Hal ini bertujuan agar kita bisa menjaga agar data sebuah object tetap baik dan valid
- Untuk mencapai ini, biasanya kita akan membuat semua field menggunakan access modifier private, sehingga tidak bisa diakses atau diubah dari luar
- Agar bisa diubah, kita akan menyediakan method untuk mengubah dan mendapatkan field tersebut

---

# Getter dan Setter

- Di Java, proses encapsulation sudah dibuat standarisasinya, dimana kita bisa menggunakan Getter dan Setter method.
- Getter adalah function yang dibuat untuk mengambil data field
- Setter ada function untuk mengubah data field

---

# Getter dan Setter Method

Tipe Data	Getter Method	Setter Method
boolean	isXxx()	setXxx(boolean value)
primitif	getXxx()	setXxx(primitif value)
Object	getXxx()	setXxx(Object value)

---

## Kode : Getter dan Setter

```
3  public class Category {  
4      private String id;  
5  
6      private boolean expensive;  
7  
8      □ public String getId() {  
9          return id;  
10     }  
11  
12     □ public void setId(String id) {  
13         this.id = id;  
14     }
```

---

## Kode : Validation di Setter

```
7
8     public String getId() {
9         return id;
10    }
11
12    public void setId(String id) {
13        if (id != null) {
14            this.id = id;
15        }
16    }
17
18    public boolean isExpensive() {
19        return expensive;
```

---

# Interface

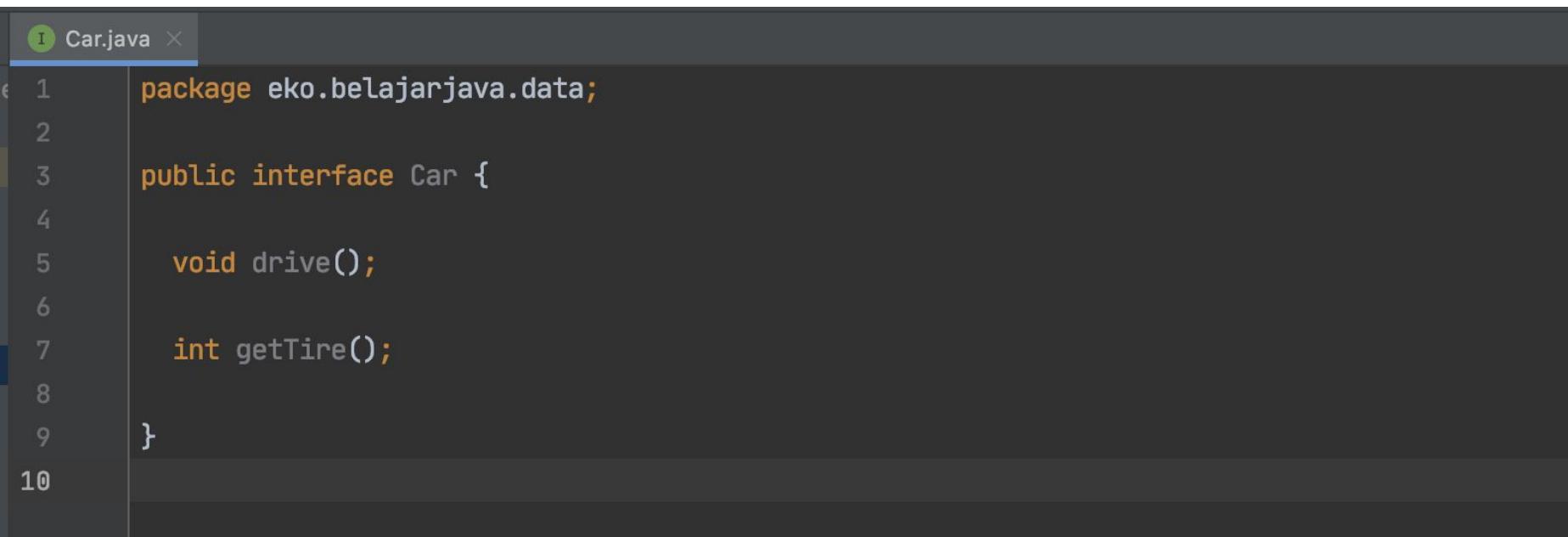
---

# Interface

- Sebelumnya kita sudah tahu bahwa abstract class bisa kita gunakan sebagai kontrak untuk class child nya.
- Namun sebenarnya yang lebih tepat untuk kontrak adalah Interface
- Jangan salah sangka bahwa Interface disini bukanlah User Interface
- Interface mirip seperti abstract class, yang membedakan adalah di Interface, semua method otomatis abstract, tidak memiliki block
- Di interface kita tidak boleh memiliki field, kita hanya boleh memiliki constant (field yang tidak bisa diubah)
- Untuk mewariskan interface, kita tidak menggunakan kata kunci extends, melainkan implements



# Kode : Membuat Interface



The image shows a screenshot of a Java code editor with a dark theme. The tab bar at the top has a green circle icon with an 'I' and the text "Car.java" followed by a close button. The code editor displays the following Java interface definition:

```
1 package eko.belajarjava.data;
2
3 public interface Car {
4
5     void drive();
6
7     int getTire();
8
9 }
10
```

# Kode : Implement Interface



The screenshot shows a Java code editor with two files open: Car.java and Avanza.java. The tab for Avanza.java is selected, indicated by a blue background. The code implements the Car interface:

```
1 package eko.belajarjava.data;
2
3 public class Avanza implements Car {
4     public void drive() {
5         System.out.println("Drive Avanza");
6     }
7
8     public int getTire() {
9         return 4;
10    }
11}
```

Annotations are present on lines 4 and 8, each with a red arrow pointing upwards and a small red circle containing a question mark.

---

# Interface Inheritance

---

# Interface Inheritance

- Sebelumnya kita sudah tahu kalo di Java, child class hanya bisa punya 1 class parent
- Namun berbeda dengan interface, sebuah child class bisa implement lebih dari 1 interface
- Bahkan interface pun bisa implement interface lain, bisa lebih dari 1. Namun jika interface ingin mewarisi interface lain, kita menggunakan kata kunci extends, bukan implements

# Kode : Interface Inheritance

```
3  ↴ public interface HasBrand {  
4  
5      String getBrand();  
6  
7  }
```

I Car.java ×

```
1  package eko.belajarjava.data;  
2  
3  ↴ public interface Car extends HasBrand {  
4  
5  ↴     void drive();
```

# Kode : Multiple Interface Inheritance

```
3  public class Avanza implements Car, IsMaintenance {  
4  
5  ①↑    public String getBrand() {  
6      return "Toyota";  
7  }  
8  
9  ①↑    public boolean isMaintenance() {  
10     return false;  
11  }  
12  
13 ①↑    public void drive() {  
14      System.out.println("Drive Avanza");  
}
```

---

# Default Method

---

# Default Method

- Sebelumnya kita tahu bahwa di interface, kita tidak bisa membuat method konkrit yang memiliki block method
- Namun sejak versi Java 8, ada fitur default method di interface.
- Fitur ini terjadi karena sulit untuk maintain kontrak interface jika sudah terlalu banyak class yang implement interface tersebut
- Ketika kita menambah satu method di interface, secara otomatis semua class yang implement akan rusak karena harus meng-override method tersebut
- Dengan menggunakan default method, kita bisa menambahkan konkrit method di interface.

---

## Kode : Default Method

```
3 1↓ public interface Car extends HasBrand {  
4  
5 1↓     void drive();  
6  
7 1↓     int getTire();  
8  
9 1↓     default boolean isBig(){  
10    |     return false;  
11 1↓ }  
12  
13 }  
14 |
```

---

# ToString Method

---

# Tostring Method

- `toString()` adalah method yang terdapat di class Object
- Method ini biasanya digunakan untuk merepresentasikan object dalam bentuk String
- Secara default, `toString()` ini akan menghasilkan :
  - namaclass + @ + hashCode
- Namun kita bisa mengubahnya jika kita mau, agar object yang kita buat lebih mudah dibaca

# Kode : Override ToString Method



```
Product.java
4
5     public String name;
6     public int price;
7
8     public Product(String name, int price) {
9         this.name = name;
10        this.price = price;
11    }
12
13 ⚡  public String toString() {
14     return "Product name:" + name + ", price:" + price;
15 }
16
17 }
```

The image shows a screenshot of a Java code editor with a dark theme. The file being edited is named 'Product.java'. The code defines a class 'Product' with two fields: 'name' (String) and 'price' (int). It has a constructor that initializes these fields. At line 13, there is a call to the 'toString()' method, which is annotated with a red lightning bolt icon, indicating it is being overridden. The implementation returns a string concatenating the product's name and price. Line numbers are visible on the left side of the code editor.

---

# Equals Method

---

# Equals Method

- Hal yang agak membingungkan di Java adalah, cara membandingkan object
- Di bahasa pemrograman lain, untuk mengecek apakah sebuah object sama, biasanya menggunakan operator ==, di Java, operator == hanya untuk mengecek data primitif
- Untuk non primitif pengecekan nya menggunakan method equals
- Dan secara default, method equals itu akan membandingkan dua buah object secara kesamaan posisi object di memory, artinya jika kita membuat 2 object yang isi fields nya sama, tetap dianggap beda oleh method equals
- Oleh karena itu, ada baiknya kita meng-override method equals milik class Object tersebut

---

# Kode : Equals Salah

```
5
6     String first = "Eko";
7     first = first + " " + "Khannedy";
8
9     String second = "Eko Khannedy";
10
11    System.out.println(first == second); // false
12
13    System.out.println(first.equals(second)); // true
14
15    }
16 }
```

# Kode : Override Equals Method

```
protected int price;

public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Product)) return false;

    Product product = (Product) o;

    if (price != product.price) return false;
    return name != null ? name.equals(product.name) : product.name == null;
}
```

# Kode : Method Objects.equals()



The screenshot shows a Java code editor with a dark theme. A file named 'Product.java' is open, and the cursor is positioned at the start of the equals() method. The code is as follows:

```
7     protected int price;  
8  
9     public boolean equals(Object o) {  
10         if (this == o) return true;  
11         if (!(o instanceof Product)) return false;  
12  
13         Product product = (Product) o;  
14         return price == product.price &&  
15             Objects.equals(name, product.name);  
16     }  
17
```

The code implements the equals() method for the Product class. It first checks if the current object is equal to the other object. If not, it checks if the other object is an instance of Product. If neither condition is met, it returns false. Finally, it uses the static method Objects.equals() to compare the names of the two products.

---

# HashCode Method

---

# HashCode Method

- Method hashCode adalah method representasi integer object kita, mirip toString yang merupakan representasi String, hashCode adalah representasi integer
- HashCode sangat bermanfaat untuk membuat struktur data unique seperti HashMap, Set, dan lain-lain, karena cukup menggunakan hashCode method untuk mendapatkan identitas unique object kita
- Secara default hashCode akan mengembalikan nilai integer sesuai data di memory, namun kita bisa mengoverride nya jika kita ingin

---

# Kontrak hashCode Method

Tidak mudah meng-override method hashCode, karena ada kontraknya :

- Sebanyak apapun hashCode dipanggil, untuk object yang sama, harus menghasilkan data integer yang sama
- Jika ada 2 object yang sama jika dibandingkan menggunakan method equals, maka nilai hashCode nya juga harus sama
- Tidak wajib hashCode berbeda jika method equals menghasilkan false, karena memang keterbatasan jumlah integer sekitar 2 miliar

---

## Kode : Override HashCode Method

```
    }

    public int hashCode() {
        int result = name != null ? name.hashCode() : 0;

        result = 31 * result + price;

        return result;
    }

    public String getName() {
        return name;
    }
```

---

## Kode : Method Objects.hash()

```
Product product = (Product) o;
return price == product.price &&
       Objects.equals(name, product.name);
}

➌ public int hashCode() {
    return Objects.hash(name, price);
}

public String getName() {
    return name;
}
```

---

# Final Class

---

# Final Class

- Sebelumnya kita pernah menggunakan kata kunci final di Java
- Jika digunakan di variable, maka variable tersebut tidak bisa berubah lagi datanya
- Final pun bisa digunakan di class, dimana jika kita menggunakan kata kunci final sebelum class, maka kita mendangkan bahwa class tersebut tidak bisa diwariskan lagi
- Secara otomatis semua class child nya akan error

---

## Kode : Final Class

```
class SocialMedia {  
    String name;  
}  
  
final class Facebook extends SocialMedia {  
}  
  
class FakeFacebook extends Facebook { // error  
}
```

---

# Final Method

---

# Final Method

- Kata kunci final juga bisa digunakan di Method
- Jika sebuah method kita tambahkan kata kunci final, maka artinya method tersebut tidak bisa di override lagi di class child nya
- Ini sangat cocok jika kita ingin mengunci implementasi dari sebuah method agar tidak bisa diubah lagi oleh class child nya

---

# Kode : Final Method

```
class Facebook extends SocialMedia {  
    final void login(String username, String password){  
    }  
}  
  
class FakeFacebook extends Facebook { // error  
    void login(String username, String password){ // error  
    }  
}
```

---

# Inner Class

---

# Inner Class

- Di Java, kita bisa membuat class di dalam class, atau disebut dengan Inner Class
- Salah satu kasus kita membuat inner class biasanya ketika kita butuh membuat beberapa class yang saling berhubungan, dimana sebuah class tidak bisa dibuat tanpa class lain
- Misal kita perlu membuat class Employee, dimana membutuhkan class Company, maka kita bisa membuat class Employee sebagai inner class Company
- Cara membuat inner class, cukup membuatnya di dalam blok class outer class nya

---

# Kode : Inner Class

```
public class Company {  
  
    private String name;  
  
    public class Employee {  
  
        private String name;  
  
        public String getName() {  
            return name;  
        }  
    }  
}
```



# Kode : Membuat Object Inner Class

```
▶ public class NestedApp {  
▶   public static void main(String[] args) {  
  
    Company company = new Company();  
    company.setName("Programmer Zaman Now");  
  
    Company.Employee employee = company.new Employee();  
    employee.setName("Eko Kurniawan");  
  
  }  
}  
|
```

---

# Mengakses Outer Class

- Keuntungan saat kita membuat inner class adalah, kemampuan untuk mengakses outer class nya
- Inner class bisa membaca semua private member yang ada di outer class nya
- Untuk mengakses object outer class nya, kita bisa menggunakan nama class outer nya diikuti dengan kata kunci this, misal Company.this
- Dan untuk mengakses super class outer class nya, kita bisa menggunakan nama class outer nya diikuti dengan kata kunci super, misal Company.super

---

# Kode : Mengakses Outer Class

```
public class Company {  
  
    private String name;  
  
    public class Employee {  
  
        private String name;  
  
        public String getCompany(){  
            return Company.this.getName();  
        }  
    }  
}
```

---

# Anonymous Class

---

# Anonymous Class

- Anonymous class atau class tanpa nama.
- Adalah kemampuan mendeklarasikan class, sekaligus meng-instansiasi object-nya secara langsung
- Anonymous class sebenarnya termasuk inner class, dimana outer class nya adalah tempat dimana kita membuat anonymous class tersebut
- Anonymous class sangat cocok ketika kita berhadapan dengan kasus membuat implementasi interface atau abstract class sederhana, tanpa harus membuat implementasi class nya

---

# Kode : Contoh Interface Sederhana

I HelloWorld.java ×

```
1 package eko.belajarjava.anonymous;  
2  
3 public interface HelloWorld {  
4  
5     void sayHello();  
6  
7     void sayHello(String name);  
8  
9 }  
10 |
```

# Kode : Contoh Anonymous Class

```
HelloWorld english = new HelloWorld() {  
    @Override  
    public void sayHello() {  
        System.out.println("Hello");  
    }  
  
    @Override  
    public void sayHello(String name) {  
        System.out.println("Hello " + name);  
    }  
};
```

---

# Static Keyword

---

# Static Keyword

- Sebelumnya kita sudah sering melihat kata kunci static, namun belum pernah kita bahas
- Dengan menggunakan kata kunci static, kita bisa membuat field, method atau class bisa diakses langsung tanpa melalui object nya
- Perlu diingat, static hanya bisa mengakses static lainnya

---

# Static Dapat Digunakan di

- Field, atau disebut class variable, artinya field tersebut bisa diakses langsung tanpa membuat object terlebih dahulu
- Method, atau disebut class method, artinya method tersebut bisa diakses langsung tanpa membuat object terlebih dahulu
- Block, static block akan otomatis dieksekusi ketika sebuah class di load
- Inner Class, artinya inner class tersebut bisa diakses secara langsung tanpa harus membuat object outer class terlebih dahulu. Static pada inner class menyebabkan kita tidak bisa mengakses lagi object outer class nya

---

## Kode : Static Field

```
public class Constant {  
  
    public static final String APPLICATION = "Belajar Java OOP";  
    public static final Integer VERSION = 1;  
  
}
```

---

# Kode : Static Method

```
public class MathUtil {  
    @  
    public static int sum(int... values) {  
        int total = 0;  
        for (var value : values) {  
            total += value;  
        }  
        return total;  
    }  
}
```

---

## Kode : Static Inner Class

```
public class Country {  
  
    private String name;  
  
    public static class City {  
  
        private String name;  
  
        public String getName() {  
            return name;  
        }  
    }  
}
```

---

# Kode : Static Block

```
public class Application {  
  
    public static final int PROCESSOR;  
  
    static {  
        PROCESSOR = Runtime.getRuntime().availableProcessors();  
    }  
}
```

---

# Kode : Memanggil Static Members

```
System.out.println(Application.PROCESSOR);

System.out.println(Constant.APPLICATION);
System.out.println(Constant.VERSION);

System.out.println(MathUtil.sum( ...values: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10));

Country.City city = new Country.City();
city.setName("Subang");

}
```



# Kode : Static Import

```
import static eko.belajarjava.statics.Constant.*;
import static eko.belajarjava.statics.Application.PROCESSOR;

public class StaticApp {
    public static void main(String[] args) {

        System.out.println(PROCESSOR);

        System.out.println(APPLICATION);
        System.out.println(VERSION);

        System.out.println(MathUtil.sum( ...values: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10));
    }
}
```

---

# Record Class

---

## Java 14 - Experimental

- Fitur ini masih versi preview dan belum stabil di versi Java 14, namun kita sudah bisa mencobanya
- Tapi perlu diingat, bahwa karena fitur ini masih experimental, artinya tidak ada jaminan di versi Java mendatang, fitur ini akan tetap ada, bisa saja dihilangkan

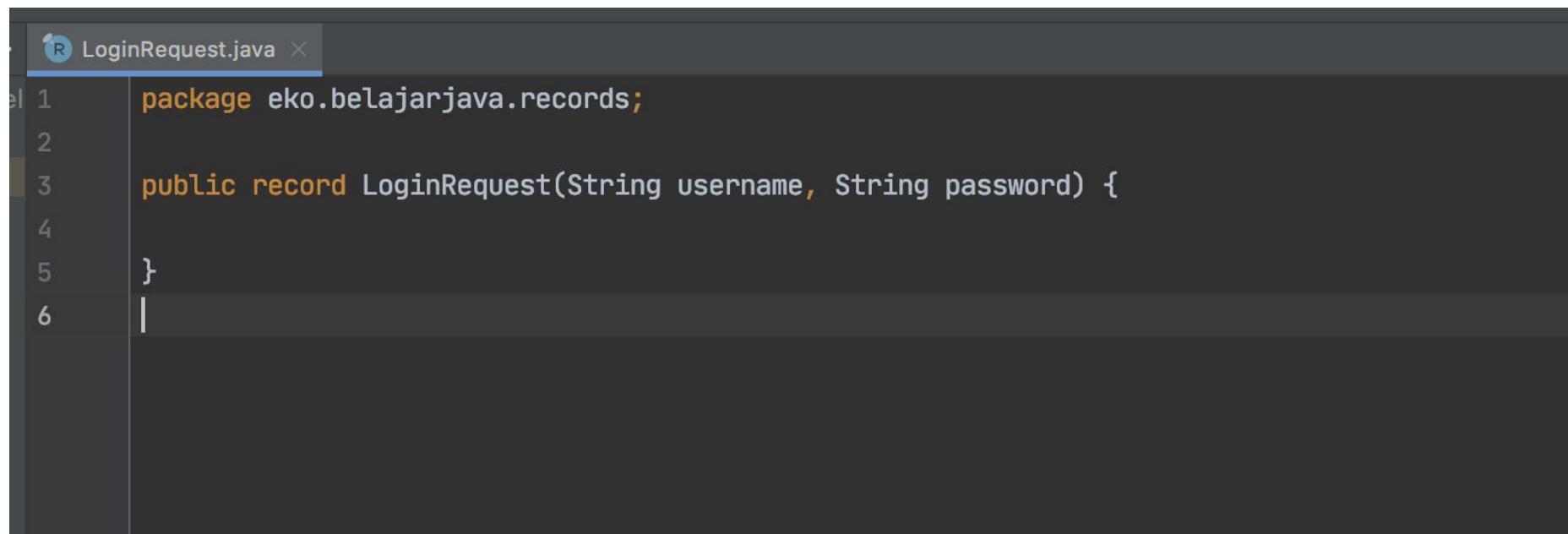
---

# Record Class

- Kadang kita sering membuat class, hanya untuk class yang berisikan data. Hanya berisi getter, equals, hashCode, dan toString method
- Record class digunakan untuk mempermudah pembuatan jenis class tersebut
- Saat kita membuat record class, secara otomatis Java akan membuatkan getter, equals, hashCode dan toString method secara otomatis, dan juga constructor secara otomatis
- Saat membuat record class, secara otomatis kita akan meng-extends class java.lang.Record, yang artinya kita tidak bisa extends class lain. Namun kita tetap bisa meng-implement interface

---

# Kode : Membuat Record Class



The screenshot shows a code editor window with a dark theme. The title bar says "LoginRequest.java". The code itself is a simple Java record definition:

```
1 package eko.belajarjava.records;
2
3 public record LoginRequest(String username, String password) {
4
5 }
```

---

# Kode : Menggunakan Record Class

```
> public class RecordApp {  
>     public static void main(String[] args) {  
  
        LoginRequest loginRequest = new LoginRequest( username: "eko", password: "rahasia");  
  
        System.out.println(loginRequest.username());  
        System.out.println(loginRequest.password());  
        System.out.println(loginRequest);  
  
    }  
}
```

---

# Record Class Constructor

- Secara default, constructor di record class akan dibuat secara otomatis, sesuai dengan definisi record class parameter
- Namun jika kita ingin melakukan sesuatu di constructor tersebut, kita bisa membuat compact constructor, yaitu constructor tanpa tanda ()
- Selain itu, kita juga bisa melakukan constructor overloading, namun ada syaratnya, yaitu constructor overloading nya harus tetap memanggil constructor utama yang secara otomatis dibuatkan di record class

---

## Kode : Record Class Constructor

```
public record LoginRequest(String username, String password) {  
    public LoginRequest {  
        System.out.println("Constructor utama dipanggil");  
    }  
    public LoginRequest(String username) {  
        this(username, password: "");  
    }  
}
```

---

# Enum Class

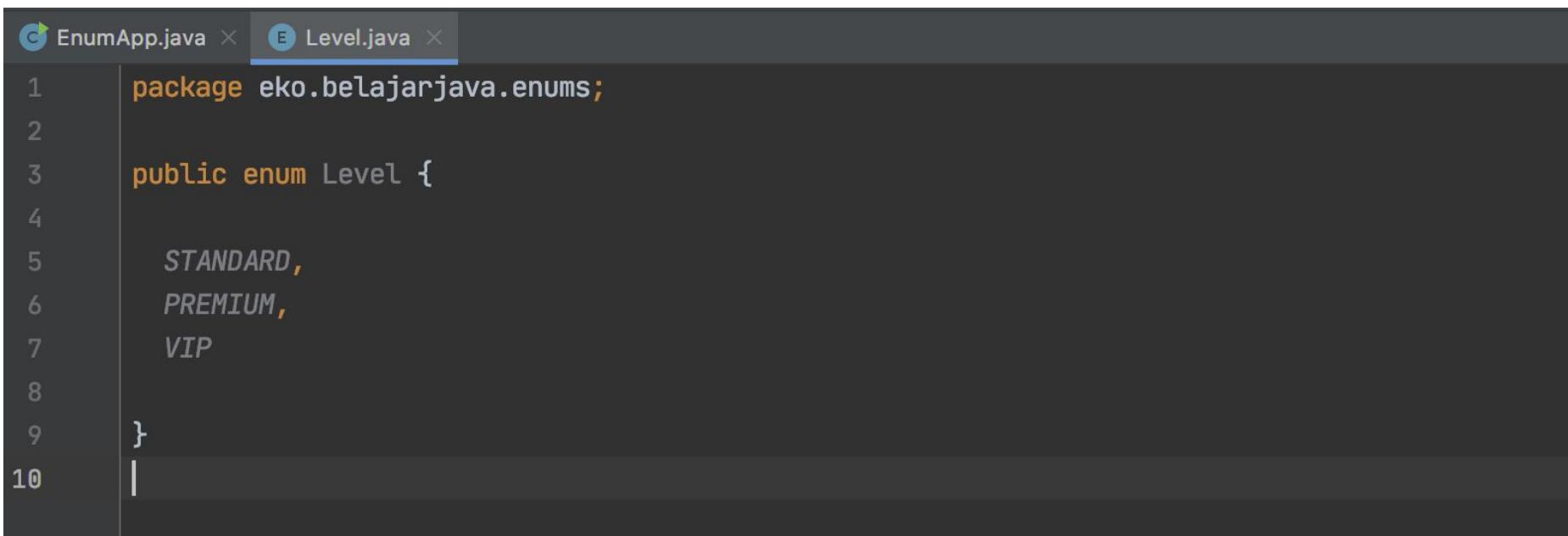
---

# Enum Class

- Saat kita membuat aplikasi, kadang kita akan bertemu dengan jenis-jenis data yang nilainya terbatas
- Misal, gender, ada male dan female, atau tipe customer, ada standard, premium atau vip, dan lain-lain
- Dalam kasus seperti ini, kita bisa menggunakan enum class, yaitu class yang berisikan nilai terbatas yang sudah ditentukan
- Saat membuat enum class, secara otomatis dia akan meng-extends class java.lang.Enum, oleh karena itu class enum tidak bisa extends class lain, namun masih tetap bisa implements interface.

---

# Kode : Membuat Enum Class



The screenshot shows a Java code editor with two tabs: "EnumApp.java" and "Level.java". The "Level.java" tab is active, displaying the following code:

```
1 package eko.belajarjava.enums;
2
3 public enum Level {
4
5     STANDARD,
6     PREMIUM,
7     VIP
8
9 }
10 |
```

The code defines an enum named "Level" with three values: "STANDARD", "PREMIUM", and "VIP". The file is numbered from 1 to 10.

---

## Kode : Menggunakan Enum

```
2
3  class Customer {
4      Level level;
5  }
6
7  public class EnumApp {
8      public static void main(String[] args) {
9          Customer customer = new Customer();
10         customer.level = Level.STANDARD;
11     }
12 }
```

---

## Enum Members

- Sama seperti class biasanya, di class enum pun kita bisa menambahkan members (field, method dan constructor)
- Khusus constructor, kita tidak bisa membuat public constructor, karena memang tujuan enum bukan untuk di instansiasi secara bebas

---

# Kode : Members di Enum

```
3 public enum Level {  
4  
5     STANDARD( description: "Standard Edition"),  
6     PREMIUM( description: "Premium Edition"),  
7     VIP( description: "VIP Edition");  
8  
9     private String description;  
10  
11    Level(String description) {  
12        this.description = description;  
13    }  
14}
```

---

# Kode : Konversi Enum ke String

```
.1  String levelString = Level.STANDARD.name();
.2
.3
.4  Level level = Level.valueOf("STANDARD");
.5
.6  Level[] values = Level.values();
.7 }
.8 H
.9
```

---

# Exception

---

# Exception

- Saat kita membuat aplikasi, kita tidak akan terhindar dengan yang namanya error
- Di Java, error direpresentasikan dengan istilah exception, dan semua direpresentasikan dalam bentuk class exception
- Kita bisa menggunakan class exception sendiri, atau menggunakan yang sudah disediakan oleh Java
- Jika kita ingin membuat exception, maka kita harus membuat class yang extends class Throwable atau turunan-turunannya

---

# Kode : Membuat Class Exception

```
public class ValidationException extends Throwable {  
  
    private String message;  
  
    public ValidationException(String message) {  
        this.message = message;  
    }  
  
    public String getMessage() {  
        return message;  
    }  
}
```

---

# Membuat Exception

- Exception biasanya terjadi di method, ketika kita membuat exception di sebuah method, maka method tersebut harus ditandai dengan kata kunci thrown diikuti dengan class exception nya.
- Jika method tersebut bisa menimbulkan lebih dari satu jenis exception, kita bisa menambah lebih dari satu class exception
- Di dalam kode program kita, untuk membuat exception kita cukup menggunakan kata kunci throw, diikuti dengan object exception nya

---

# Kode : Membuat Exception

```
public class ValidationUtil {  
    @  
    public static void validate(LoginRequest loginRequest) throws ValidationException {  
        if (loginRequest.username() == null) {  
            throw new ValidationException("Username tidak boleh null");  
        } else if (loginRequest.username().isBlank()) {  
            throw new ValidationException("Username tidak boleh kosong");  
        } else if (loginRequest.password() == null) {  
            throw new ValidationException("Password tidak boleh null");  
        } else if (loginRequest.password().isBlank()) {  
            throw new ValidationException("Password tidak boleh kosong");  
        }  
    }  
}
```

---

# Try Catch

- Saat kita memanggil sebuah function yang bisa menyebabkan exception, maka kita wajib menggunakan try-catch expression di Java
- Ini gunanya agar kita bisa menangkap exception yang terjadi, karena jika tidak ditangkap, lalu terjadi exception, maka secara otomatis program kita akan berhenti
- Cara menggunakan try-catch expression di java sangat mudah, di block try, kita tinggal panggil method yang bisa menyebabkan exception, dan di block catch, kita bisa melakukan sesuatu jika terjadi exception

---

## Kode : Try Catch

```
LoginRequest loginRequest = new LoginRequest( username: null, password: null);

try {
    ValidationUtil.validate(loginRequest);
} catch (ValidationException e) {
    System.out.println("Terjadi Error Dengan Pesan : " + e.getMessage());
}

}
```

---

## Kode : Multiple Try Catch (1)

```
LoginRequest loginRequest = new LoginRequest( username: null, password: null);

try {
    ValidationUtil.validate(loginRequest);
} catch (ValidationException e) {
    System.out.println("Terjadi Error Dengan Pesan : " + e.getMessage());
} catch (NullPointerException e) {
    System.out.println("Terjadi Error Dengan Pesan : " + e.getMessage());
}
```

## Kode : Multiple Try Catch (2)

```
LoginRequest loginRequest = new LoginRequest( username: null, password: null);

try {
    ValidationUtil.validate(loginRequest);
} catch (ValidationException | NullPointerException e) {
    System.out.println("Terjadi Error Dengan Pesan : " + e.getMessage());
}

}
```

---

# Finally Keyword

- Dalam try-catch, kita bisa menambahkan block finally
- Block finally ini adalah block dimana akan selalu dieksekusi baik terjadi exception ataupun tidak
- Ini sangat cocok ketika kita ingin melakukan sesuatu, tidak peduli sukses ataupun gagal, misal di block try kita ingin membaca file, di block catch kita akan tangkap jika terjadi error, dan di block finally error ataupun sukses membaca file, kita wajib menutup koneksi ke file tersebut, biar tidak menggantung di memory

---

## Kode : Finally

```
LoginRequest loginRequest = new LoginRequest( username: null, password: null);

try {
    ValidationUtil.validate(loginRequest);
} catch (ValidationException | NullPointerException e) {
    System.out.println("Terjadi Error Dengan Pesan : " + e.getMessage());
} finally {
    System.out.println("Error Gak Error, Tetap Di Panggil");
}
```

---

# Runtime Exception

---

# Jenis Exception

Secara garis besar, di Java, exception dibagi menjadi 3 jenis

- Checked Exception, yaitu exception yang wajib di try catch, seperti yang sudah kita bahas sebelumnya
- Runtime Exception, dan
- Error (yang akan kita bahas di materi selanjutnya)

---

# Runtime Exception

- Runtime exception adalah jenis exception yang tidak wajib di tangkap menggunakan try catch
- Kompiler Java tidak akan protes walaupun kita tidak menggunakan try catch ketika kita memanggil method yang bisa menyebabkan runtime exception
- Untuk membuat class runtime exception, kita wajib mengextends class RuntimeException
- Ada banyak di Java yang merupakan runtime exception, seperti NullPointerException, IllegalArgumentException, dan lain-lain

---

# Kode : Membuat Runtime Exception

```
package eko.belajarjava.error;

public class BlankException extends RuntimeException{

    public BlankException(String message) {
        super(message);
    }
}
```

---

## Kode : Method Dengan Runtime Exception

```
@
public static void validateRuntime(LoginRequest loginRequest) {
    if (loginRequest.username() == null) {
        throw new BlankException("Username tidak boleh null");
    } else if (loginRequest.username().isBlank()) {
        throw new BlankException("Username tidak boleh kosong");
    } else if (loginRequest.password() == null) {
        throw new BlankException("Password tidak boleh null");
    } else if (loginRequest.password().isBlank()) {
        throw new BlankException("Password tidak boleh kosong");
    }
}
```



# Kode : Tanpa Try Catch

```
> public class ValidationApp {  
>     public static void main(String[] args) {  
  
        LoginRequest loginRequest = new LoginRequest( username: null, password: null);  
        ValidationUtil.validateRuntime(loginRequest);  
  
    }  
}
```

---

## Perlu Diperhatikan

- Walaupun runtime exception tidak wajib di try-catch, tapi ada baiknya kita tetap melakukan try-catch
- Karena jika terjadi runtime exception, yang ditakutkan adalah program kita berhenti

---

# Error

---

# Error

- Error adalah jenis exception yang terakhir
- Error adalah sebuah class di Java, yang tidak direkomendasikan untuk di try-catch
- Biasanya error terjadi ketika ada masalah serius, dan sangat tidak direkomendasikan untuk di try catch
- Artinya, direkomendasikan untuk mematikan aplikasi
- Contoh, misal jika diawal aplikasi kita tidak bisa terkoneksi ke database, direkomendasikan untuk membuat exception jenis Error, dan menghentikan aplikasi

---

# Kode : Membuat Error

```
package eko.belajarjava.error;

public class DatabaseError extends Error{

    public DatabaseError(String message) {
        super(message);
    }
}
```

# Kode : Terjadi Error

```
▶  public class DatabaseApp {  
▶    public static void main(String[] args) {  
▶      connectDatabase(username: "admin", password: null);  
▶    }  
▶  
▶    public static void connectDatabase(String username, String password) {  
▶      if (username == null || password == null) {  
▶        throw new DatabaseError("Tidak bisa koneksi ke database");  
▶      }  
▶    }  
▶  }
```

---

# StackTraceElement Class

---

# StackTraceElement Class

- Throwable memiliki method yang bernama getStackTrace(), dimana menghasilkan Array dari StackTraceElement object
- StackTraceElement berisikan informasi tentang, class, file bahkan baris lokasi terjadinya error
- Class StackTraceElement ini sangat penting untuk menelusuri lokasi kejadian error yang terjadi
- Cara yang paling mudah, kita bisa memanggil method printStackTrace() class Throwable, untuk memprint ke console detail error StackTraceElement nya

---

# Kode : StackTraceElement

```
▶ public static void main(String[] args) {  
    try {  
        String[] names = {  
            "Eko", "Kurniawan", "Khannedy"  
        };  
        System.out.println(names[100]);  
    }catch (Throwable throwable){  
        StackTraceElement[] stackTraces = throwable.getStackTrace();  
  
        throwable.printStackTrace();  
    }  
}
```

---

# Kode : Multiple StackTraceElement

```
public static void sampleError() throws Throwable {
    try {
        String[] names = {
            "Eko", "Kurniawan", "Khannedy"
        };
        System.out.println(names[100]);
    } catch (Throwable throwable) {
        throw new Throwable(throwable);
    }
}
```

---

# Try with Resource

---

# Try with Resource

- Di Java 7, terdapat fitur baru yang bernama try with resource
- Try with resource adalah sebuah mekanisme agar kita lebih mudah menggunakan resource (yang wajib di close) dalam block try
- Jika kita ingin menggunakan fitur ini, kita wajib menggunakan interface AutoCloseable

# Kode : Manual Close Resource

```
BufferedReader reader = null;
try {
    reader = new BufferedReader(
        new FileReader( fileName: "sample")
    );
    while (true) {
        String text = reader.readLine();
        if (text == null) {
            break;
        }
        System.out.println(text);
    }
} catch (IOException exception) {
    exception.printStackTrace();
} finally {
    if (reader != null) {
        try {
            reader.close();
        } catch (IOException exception) {
            exception.printStackTrace();
        }
    }
}
```



# Kode : Try with Resource

```
try (BufferedReader reader = new BufferedReader(new FileReader(fileName: "sample"))) {  
    while (true) {  
        String text = reader.readLine();  
        if (text == null) {  
            break;  
        }  
        System.out.println(text);  
    }  
} catch (IOException exception) {  
    exception.printStackTrace();  
}
```

---

# Annotation

---

# Annotation

- Annotation adalah menambahkan metadata ke kode program yang kita buat
- Tidak semua orang membutuhkan Annotation, biasanya Annotation digunakan saat kita ingin membuat library / framework
- Annotation sendiri bisa diakses menggunakan Reflection, yang akan kita bahas nanti
- Untuk membuat annotation, kita bisa menggunakan kata kunci @interface
- Annotation hanya bisa memiliki method dengan tipe data sederhana, dan bisa memiliki default value

---

# Attribute Annotation

Attribute	Keterangan
@Target	Memberitahu annotation bisa digunakan di mana? Class, method, field, dan lain-lain
@Retention	Memberitahu annotation apakah disimpan di hasil kompilasi, dan apakah bisa dibaca oleh reflection?

---

# Kode : Membuat Annotation

```
import java.lang.annotation.*;

@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = {ElementType.TYPE})
public @interface Fancy {

    String name();

    String[] tags() default {};

}
```

---

# Kode : Menggunakan Annotation

```
package eko.belajarjava.annotation;

@Fancy(name = "Eko", tags = {"app", "java"})
public class Application {

}
```

---

# Predefined Annotation

Java juga sudah memiliki annotation bawaan, seperti :

- @Override, untuk menandai bahwa method yang meng-override method parent class nya
- @Deprecated, untuk menandai bahwa method tersebut tidak direkomendasikan lagi untuk digunakan
- @FunctionallInterface, untuk menandai bahwa class tersebut bisa dibuat sebagai lambda expression
- dan lain-lain

---

# Reflection

---

# Reflection

- Reflection adalah kemampuan melihat struktur aplikasi kita pada saat berjalan
- Reflection biasanya sangat berguna saat kita ingin membuat library ataupun framework, sehingga bisa meng-otomatisasi pekerjaan
- Untuk mengakses reflection class dari sebuah object, kita bisa menggunakan method getClass() atau NamaClass.class

---

## Kode : Annotation di Field

```
public class CreateUserRequest {  
  
    @NotBlank  
    private String username;  
  
    @NotBlank  
    private String password;  
  
    public String getUsername() {  
        return username;  
    }  
}
```



# Kode : Validasi Menggunakan Reflection

```
public static void validateRequest(CreateUserRequest request) throws IllegalAccessException, ValidationException {
    Field[] fields = request.getClass().getDeclaredFields();
    for (Field field : fields) {
        if (field.getAnnotation(NotBlank.class) != null) {
            field.setAccessible(true);
            String value = (String) field.get(request);
            if (value == null || value.isBlank()) {
                throw new ValidationException("Field " + field.getName() + " is blank");
            }
        }
    }
}
```

---

## Perlu Diperhatikan

- Reflection adalah materi yang sangat panjang
- Oleh karena itu materi Java Reflection akan dibuatkan course terpisah

---

# Materi Selanjutnya

---

# Materi Selanjutnya

- Java Classes
- Java Generic
- Java Collection