
Java Generic

Sebelum Belajar Materi Ini

- Java Dasar
- Java Object Oriented Programming
- Java Standard Classes

Agenda

- Pengenalan Generic
- Generic Class
- Generic Method
- Invariant, Covariant dan Contravariant
- Wildcard
- Dan lain-lain

Pengenalan Generic

Pengenalan Generic

- Generic adalah kemampuan menambahkan parameter type saat membuat class atau method
- Berbeda dengan tipe data yang biasa kita gunakan di class di function, generic memungkinkan kita bisa mengubah-ubah bentuk tipe data sesuai dengan yang kita mau.

Manfaat Generic

- Pengecekan ketika proses kompilasi
- Tidak perlu manual menggunakan pengecekan tipe data dan konversi tipe data
- Memudahkan programmer membuat kode program yang generic sehingga bisa digunakan oleh berbagai tipe data

Kode : Bukan Generic

```
    Data dataString = new Data();
    dataString.setData("Eko");
    String value = (String) dataString.getData();
}

public static class Data {

    private Object data;

    public Object getData() {
        return data;
    }
}
```



Kode : Generic

```
    Data<String> dataString = new Data<String>();
    dataString.setData("Eko");
    String value = dataString.getData();
}

public static class Data<T> {

    private T data;

    public T getData() {
        return data;
    }
}
```

Generic Class

Generic Class

- Generic class adalah class atau interface yang memiliki parameter type
- Tidak ada ketentuan dalam pembuatan generic parameter type, namun biasanya kebanyakan orang menggunakan 1 karakter sebagai generic parameter type
- Nama generic parameter type yang biasa digunakan adalah :
 - E - Element (biasa digunakan di collection atau struktur data)
 - K - Key
 - N - Number
 - T - Type
 - V - Value
 - S,U,V etc. - 2nd, 3rd, 4th types

Kode : Generic Class

```
public class MyData<T> {  
    private T data;  
  
    public MyData(T data) {  
        this.data = data;  
    }  
  
    public T getData() {  
        return data;  
    }  
  
    public void setData(T data) {
```

Kode : Membuat Generic Object

```
MyData<String> myDataString = new MyData<String>( data: "Eko");
MyData<Integer> myDataInteger = new MyData<>( data: 100);
var myDataBoolean = new MyData<Boolean>( data: true);

System.out.println(myDataString.getData());
System.out.println(myDataInteger.getData());
System.out.println(myDataBoolean.getData());

}
```

Multiple Parameter Type

- Parameter type di Generic class boleh lebih dari satu
- Namun harus menggunakan nama type berbeda
- Ini sangat berguna ketika kita ingin membuat generic parameter type yang banyak



Kode : Multiple Parameter Type

```
public class Pair<T, U> {  
  
    private T first;  
    private U second;  
  
    public Pair(T first, U second) {  
        this.first = first;  
        this.second = second;  
    }  
  
    public T getFirst() { return first; }  
}
```



Kode : Multiple Parameter Type Object

```
▶  public static void main(String[] args) {  
    Pair<String, Integer> pair = new Pair<String, Integer>("Eko", 20);  
  
    System.out.println(pair.getFirst());  
    System.out.println(pair.getSecond());  
}  
}
```

Generic Method

Generic Method

- Generic parameter type tidak hanya bisa digunakan pada class atau interface
- Kita juga bisa menggunakan generic parameter type di method
- Generic parameter type yang kita deklarasikan di method, hanya bisa diakses di method tersebut, tidak bisa digunakan di luar method
- Ini cocok jika kita ingin membuat generic method, tanpa harus mengubah deklarasi class



Kode : Generic Method

```
public class ArrayHelper {  
    @  
    public static <T> int count(T[] array) {  
        return array.length;  
    }  
}
```

Kode : Menggunakan Generic Method

```
String[] names = {"Eko", "Kurniawan", "Khannedy"};
Integer[] values = {1, 2, 3, 4, 5};

System.out.println(ArrayHelper.<String>count(names));
System.out.println(ArrayHelper.count(values));

}
```

Invariant



Invariant

- Secara default, saat kita membuat generic parameter type, sifat parameter tersebut adalah invariant
- Invariant artinya tidak boleh di substitusi dengan subtype (child) atau supertype (parent)
- Artinya saat kita membuat object Contoh<String>, maka tidak sama dengan Contoh<Object>, begitupun sebaliknya, saat membuat object Contoh<Object>, maka tidak sama dengan Contoh<String>

Kode Program : Invariant

```
MyData<String> dataString = new MyData<>( data: "Eko");  
MyData<Object> dataObject = dataString; // error
```

```
MyData<Object> data = new MyData<>( data: 100);  
MyData<Integer> dataInteger = data; // error
```

```
}
```

```
}
```

```
|
```

Covariant

Covariant

- Covariant artinya kita bisa melakukan substitusi subtype (child) dengan supertype (parent)
- Caranya agar generic object kita menjadi covariant adalah dengan menggunakan kata kunci (? extends ParentClass)
- Artinya saat kita membuat object Contoh<String>, maka bisa disubstitusi menjadi Contoh<? extends Object>
- Covariant adalah read-only, jadi kita tidak bisa mengubah data generic nya

Kode : Covariant

```
    MyData<String> data = new MyData<>( data: "Eko");
    process(data);
}

@  public static void process(MyData<? extends Object> data){
    Object object = data.getData();
    data.setData("Eko"); // error
}
```

Contravariant

Contravariant

- Contravariant artinya kita bisa melakukan substitusi supertype (parent) dengan subtype (child)
- Caranya agar generic object kita menjadi contravariant adalah dengan menggunakan kata kunci (? super SubClass)
- Artinya saat kita membuat object Contoh<Object>, maka bisa disubstitusi menjadi Contoh<? super String>
- Contravariant adalah bisa write dan read, namun perlu berhati-hati ketika melakukan read, terutama jika sampai parent nya punya banyak child

Kode Program : Contravariant

```
MyData<Object> objectMyData = new MyData<>( data: "Eko");
MyData<? super String> myData = objectMyData;

}

public static void process(MyData<? super String> myData){
    myData.setData("Eko");
}
```

Bounded Type Parameter

Bounded Type Parameter

- Kadang kita ingin membatasi data yang boleh digunakan di generic parameter type
- Kita bisa menambahkan constraint di generic parameter type dengan menyebutkan tipe yang diperbolehkan
- Secara otomatis, type data yang bisa digunakan adalah type yang sudah kita sebutkan, atau class-class turunannya
- Secara default, constraint type untuk generic parameter type adalah Object, sehingga semua tipe data bisa digunakan

Kode : Bounded Type Parameter

```
public static void main(String[] args) {
    NumberData<Integer> integerNumberData = new NumberData<>( data: 100);

    // error
    NumberData<String> stringNumberData = new NumberData<String>( data: "Eko");
}

public static class NumberData<T extends Number> {
    private T data;

    public NumberData(T data) {
        this.data = data;
    }
}
```

Multiple Bounded Type Parameter

- Kadang kita ingin membatasi tipe data dengan beberapa jenis tipe data di generic parameter type
- Kita bisa menambahkan beberapa bounded type parameter dengan karakter & setelah bounded type pertama
- Jika ingin menambahkan lagi, cukup gunakan karakter & diikuti bounded type nya lagi



Kode : Inheritance

```
public static interface CanSayHello {  
    void sayHello(String name);  
}  
  
public static abstract class Employee {  
}  
  
public static class Manager extends Employee {  
}  
  
public static class VicePresident extends Employee implements CanSayHello {
```

Kode : Multiple Bounded Type Parameter

```
var manager = new Data<Manager>(new Manager());
var vp = new Data<VicePresident>(new VicePresident());
}

public static class Data<T extends Employee & CanSayHello> {
    private T data;

    public Data(T data) {
        this.data = data;
    }

    public T getData() {
```

Wildcard

Wildcard

- Kadang ada kasus kita tidak peduli dengan generic parameter type pada object
- Misal kita hanya ingin mem-print data T, tidak peduli tipe apapun
- Jika kita mengalami kasus seperti ini, kita bisa menggunakan wildcard
- Wildcard bisa dibuat dengan mengganti generic parameter type dengan karakter ?



Kode : Wildcard

```
public static void main(String[] args) {
    printLength(new MyData<>( data: 100));
    printLength(new MyData<>( data: "Eko"));
    printLength(new MyData<>( data: true));
}

public static void printLength(MyData<?> data){
    System.out.println(data.getData());
}
```

Type Erasure

Type Erasure

- Type erasure adalah proses pengecekan generic pada saat compile time, dan menghiraukan pengecekan pada saat runtime
- Type erasure menjadikan informasi generic yang kita buat akan hilang ketika kode program kita telah di compile menjadi binary file
- Compiler akan mengubah generic parameter type menjadi tipe Object di Java

Kode : Type Erasure

```
public static class Data<T> {  
    private T data;  
  
    public Data(T data) {  
        this.data = data;  
    }  
  
    public T getData() {  
        return data;  
    }  
  
    public void setData(T data) {
```

```
public static class Data {  
    private Object data;  
  
    public Data(Object data) {  
        this.data = data;  
    }  
  
    public Object getData() {  
        return data;  
    }  
  
    public void setData(Object data) {
```

Problem Type Erasure

- Karena informasi generic hilang ketika sudah menjadi binary file
- Oleh karena itu, konversi tipe data generic akan berbahaya jika dilakukan secara tidak bijak

Kode : Problem Type Erasure

```
Data stringData = new Data<>( data: "Eko");

Data<Integer> integerData = (Data<Integer>) stringData;
Integer integer = integerData.getData(); // error
}

public static class Data<T> {
    private T data;

    public Data(T data) {
        this.data = data;
    }
}
```

Comparable Interface

Comparable

- Sebelumnya kita sudah tahu bahwa operator perbandingan object menggunakan method equals
- Bagaimana dengan operator perbandingan lainnya? Seperti kurang dari atau lebih dari?
- Operator perbandingan tersebut bisa kita lakukan, jika object kita mewariskan interface generic Comparable
- Ini banyak sekali digunakan seperti untuk proses pengurutan data misalnya
- <https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/lang/Comparable.html>

Kode : Comparable

```
public class Person implements Comparable<Person> {

    private String name;
    private String address;

    @Override
    public int compareTo(Person o) {
        return this.name.compareTo(o.name);
    }

    public Person(String name, String address) {
        this.name = name;
    }
}
```



Kode : Menggunakan Comparable

```
Person[] people = {  
    new Person( name: "Eko", address: "Indonesia"),  
    new Person( name: "Budi", address: "Indonesia"),  
    new Person( name: "Joko", address: "Indonesia")  
};  
  
Arrays.sort(people);  
  
System.out.println(Arrays.toString(people));  
}  
}
```

Comparator Interface

Comparator Interface

- Jika kita ingin mengurutkan class yang kita gunakan, cukup mudah tinggal implement interface Comparable
- Namun bagaimana jika class tersebut milik orang lain? Tidak bisa kita ubah?
- Maka kita bisa menggunakan interface generic yang bernama Comparator
- <https://docs.oracle.com/en/java/javase/14/docs/api/java.base/java/util/Comparator.html>

Kode : Menggunakan Comparator

```
Comparator<Person> comparator = new Comparator<Person>() {  
    @Override  
    public int compare(Person o1, Person o2) {  
        return o1.getAddress().compareTo(o2.getAddress());  
    }  
};  
  
Arrays.sort(people, comparator);  
  
System.out.println(Arrays.toString(people));
```

Materi Selanjutnya

Materi Selanjutnya

- Java Collection