

POLYMORPHISM

Fundamen Pengembangan Aplikasi

Pertemuan 13



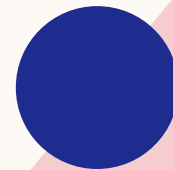
UNIVERSITAS
ISLAM
INDONESIA

TOPIK MATERI

Polimorfisme

Casting (Upcasting & Downcasting)

instanceof



OBJECT ORIENTED PROGRAMMING (OOP)

Empat konsep dasar OOP:

1. Encapsulation
2. Inheritance
3. Abstraction
4. **Polymorphism**



POLYMORPHISM

PENGERTIAN POLIMORFISME

- *Poli & Morphos* (Yunani) = **banyak bentuk**
- *Polimorfisme* (*polymorphism*) adalah sebuah prinsip dalam biologi di mana organisme atau spesies dapat memiliki **banyak bentuk** atau **tahapan** (stages).
- **Polymorphism** di Java:
 - *Overloading* dan *Overriding*
 - Sebuah prinsip di mana **class** dapat **memiliki banyak** “bentuk” method yang berbeda-beda meskipun **namanya sama**.
 - “**Bentuk**” di sini dapat diartikan: isinya berbeda, parameternya berbeda, dan tipe datanya berbeda.
 - Memungkinkan *object subclass* diperlakukan sebagai sebuah *object superclass*, namun akan melakukan tindakan sesuai dengan *object subclass* tersebut (*reference variable*).

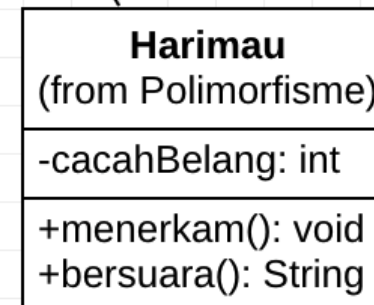
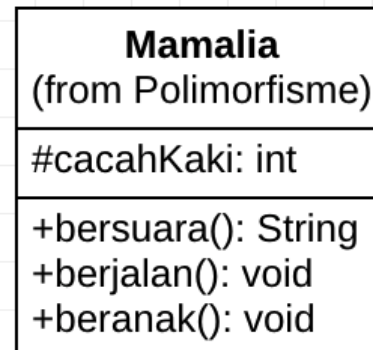
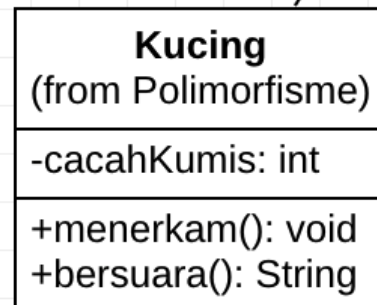
IMPLEMENTASI POLIMORFISME

- Polymorphism dapat diwujudkan melalui **relasi pewarisan**.
- Dalam Java, sebuah **rujukan** yang dideklarasikan untuk *objek suatu kelas*, dapat digunakan untuk *merujuk ke objek kelas yang lain*, yang *dihubungkan melalui hubungan pewarisan*.
- Contoh:
Kelas **Kucing** merupakan pewarisan dari kelas **Mamalia**.
Suatu **variabel referensi** yang merujuk ke objek kelas **Mamalia** dapat digunakan untuk merujuk ke objek kelas **Kucing**

```
Kucing k = new Kucing();  
Mamalia m = k;
```
- **m** dan **k** adalah *variabel referensi* bertipe kelas yang berbeda dan hal ini dapat dilakukan.

Perhatikan method
bersuara()

Mamalia punya
banyak bentuk
→ Polimorfism



Kelas **Mamalia**
dapat merujuk ke
instance dari kelas
Kucing atau kelas
Harimau

OVERRIDING

- Terdapat method **bersuara()** pada superclass maupun pada subclass-nya. Konsep ini disebut dengan *Overriding*.
- Terdapat *perilaku subclass yang lebih spesifik* dari pada superclass-nya.
- Dilakukan dengan cara **mendeklarasikan kembali method** milik *parent class* di *subclass*. Deklarasi method pada subclass **harus sama** dengan yang terdapat di superclass, yaitu pada:
 - Nama
 - Return type
 - Daftar parameter (cacah, tipe, dan urutan)
- Method superclass disebut *overriden* method
- Method subclass disebut *overriding* method.


```
class Mamalia {  
    protected int cacahKaki;  
  
    public String bersuara() {  
        return ("bersuara");  
    }  
}
```

```
class Harimau extends Mamalia {  
    private int cacahBelang;  
  
    @Override  
    public String bersuara() {  
        return ("Auuuumhgrrrrrr...");  
    }  
}
```

```
class Kucing extends Mamalia{  
    private int cacahKumis;  
  
    @Override  
    public String bersuara() {  
        return ("Meooong...");  
    }  
}
```

TES POLIMORFISME

```
public class TesPolimorfisme {  
    public static void main(String[] args) {  
        String kataTes[] = new String[3];  
        Mamalia[] m = new Mamalia[3];  
        m[0] = new Mamalia();  
        m[1] = new Kucing();  
        m[2] = new Harimau();  
  
        for (int i=0; i<3; i++) {  
            kataTes[i] = m[i].bersuara();  
        }  
    }  
}
```

Bukti polimorfisme = satu method akan memberikan aksi yang berbeda-beda.

✓ VARIABLES

✓ Local

args: String[0]@8

✓ kataTes: String[3]@9

> 0: "Bersuara"

> 1: "Meoonggg"

> 2: "Aummmmhgrrrrrrrr"

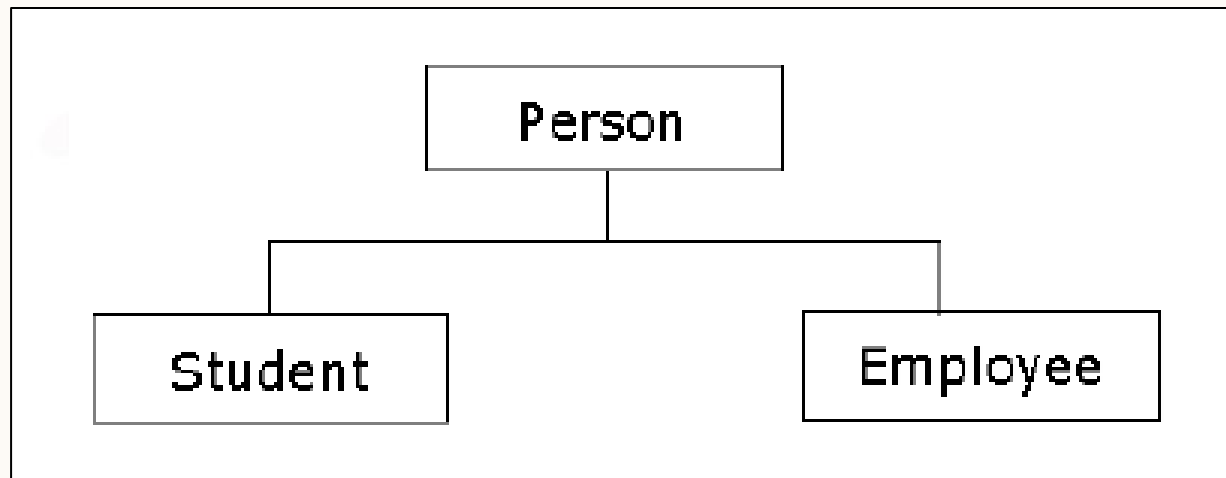
```
Mamalia[] m = new Mamalia[3]; m = Mamalia[3]@10  
m[0] = new Mamalia(); m = Mamalia[3]@10  
m[1] = new Kucing(); m = Mamalia[3]@10  
m[2] = new Harimau(); m = Mamalia[3]@10
```



```
for (int i=0; i<3; i++) {  
    kataTes[i] = m[i].bersuara();  
}
```

CONTOH POLIMORFISME

- Parent class: **Person**
- Child classes: **Employee** dan **Student**
- Hierarki kelas:



CONTOH POLIMORFISME

- Perhatikan method **getName** pada *superclass* **Person** yang di-*override* pada *subclass* **Student** dan **Employee**.

```
1 package contoh;
2
3 public class Employee extends Person {
4     public Employee(String name){
5         this.name = name;
6     }
7
8     @Override
9     public String getName(){
10         return "Employee name: "+name;
11     }
12 }
```

[Go to Super Implementation](#)

Go to super method 'contoh.Person.getName'()

```
1 package contoh;
2
3 public class Person {
4     protected String name;
5
6     public String getName(){
7         return "His name is unknown";
8     }
9 }
```

```
1 package contoh;
2
3 public class Student extends Person {
4     public Student(String name){
5         this.name = name;
6     }
7
8     @Override
9     public String getName(){
10         return "Student name: "+name;
11     }
12 }
```

[Go to Super Implementation](#)

Go to super method 'contoh.Person.getName'()

CONTOH POLIMORFISME

- Dapat dibuat sebuah **reference variabel** yang bertipe **superclass** (**Person**) pada sebuah **object subclass** (**Student**).
- **Polimorfisme** dalam Java memungkinkan suatu **reference variable** untuk bertindak (melakukan *method*) sesuai dengan object yang dipegangnya

```
1  package contoh;
2
3  public class TesContoh {
4      Run | Debug
5      public static void main(String[] args) {
6          Student st = new Student(name:"Sinta");
7
8          /*
9           * Objek ref bertipe Person
10          * Objek ref kemudian merujuk ke objek st yang bertipe Student
11          */
12          Person ref = st;
13
14          /*
15           * Objek ref dapat memanggil method getName() milik objek st
16          */
17          System.out.println(ref.getName());
18      }
19  }
```

```
CODING JAVA\00 - Kuliah FPA\P13\P13\bin' 'contoh.TesContoh'
Student name: Sinta
```

CONTOH POLIMORFISME

- Jika variable **ref** *di-assign* ke objek **Employee** dan *method* **getName()** diakses, yang terjadi adalah pemanggilan **getName()** milik **Employee** (bukan **getName()** milik **Student**)

```
public class TesContoh {  
    Run | Debug  
    public static void main(String[] args) {  
        Student st = new Student(name:"Sinta");  
        Employee em = new Employee(name:"Agus");  
  
        /* ...  
        Person ref = st;  
  
        /* ...  
        System.out.println(ref.getName());  
  
        ref = em; //Objek ref merujuk ke objek em yang bertipe Employee  
        System.out.println(ref.getName()); //getName() milik Employee yang dipanggil  
    }  
}
```

```
Student name: Sinta  
Employee name: Agus  
PS D:\CODING JAVA\00 - Kuliah FPA\P13\P13>
```

CONTOH POLIMORFISME (2)

- Contoh lain implementasi *polymorphism* adalah dengan melakukan **passing** sebuah *reference variable* sebagai **parameter input** sebuah *method*.
- Misalkan kita tambahkan sebuah *static method* **printInformation** yang menggunakan **type Person** sebagai **parameter**

```
public static void printInformation(Person p){  
    /*  
    * Method ini akan memanggil method getName() -  
    * sesuai objek yang menjadi parameter input -  
    * dari pemanggilan method printInformation()  
    */  
    System.out.println(p.getName());  
}
```

CONTOH POLIMORFISME (2)

- Kita dapat menggunakan objek dengan tipe **Employee** ataupun **Student** untuk dijadikan *input method* **printInformation**.
- Mengapa? Karena **Employee** dan **Student** adalah **Person**

```
Run | Debug
public static void main(String[] args) {
    Student st = new Student(name:"Sinta");
    Employee em = new Employee(name:"Agus");
    printInformation(st);
    printInformation(em);
}
```

```
Student name: Sinta
Employee name: Agus
PS D:\CODING JAVA\00 - Kuliah FPA\P13\P13>
```


KEUNTUNGAN POLIMORFISME

Simplicity

- Jika kita memerlukan menulis **suatu kode** yang dapat menggunakan berbagai **jenis tipe**, **kode** hanya **cukup berinteraksi** dengan *base class* (*parents*) dan dapat mengabaikan detail yang lebih spesifik yang berada pada *subclass*.
- Memudahkan penulisan program dan memudahkan pihak lain

Extensibility

- *Subclass* yang lain dapat dibuat sebagai **tipe baru** dan objek dari *class* baru tersebut masih **kompatibel** dengan **kode yang sudah ada**



CASTING

CASTING

- *Casting* merupakan proses **konversi**/mengubah tipe data dari sebuah **data/variabel primitif** (int, float, double, dll) atau **objek**.
- Pada variabel primitif, setelah dilakukan konversi maka nilainya akan disimpan dan tidak dapat dikembalikan lagi seperti semula (*irreversible*).

```
public class CastingPrimitif {  
    Run | Debug  
    public static  
        double angka = 4.37;  
        int hasilKonversi = (int) angka;  
  
        System.out.println(angka+" dikonversi ke int: "+hasilKonversi);  
        System.out.println("Dikembalikan ke double: "+(double) hasilKonversi);  
}  
}
```

Konversi dari **double** menjadi **int**

Konversi dari **int** menjadi **double**

```
4.37 dikonversi ke int: 4  
Dikembalikan ke double: 4.0  
PS D:\CODING JAVA\00 - Kuliah FPA\P13\P13>
```

Hasil konversi *irreversible*

CASTING OBJEK

- Sebuah *reference variable* **HANYA** merujuk pada objek dan **TIDAK BERISI** objek itu sendiri.
- Ketika dilakukan *casting*, **objek tidak berubah**. Objek *hanya* diberi “**label**” yang berbeda.
- Contoh:
 - Kita membuat objek **Kucing** (*subclass*) yang kemudian di-*upcast* ke **Mamalia** (*superclass*).
 - Objek **Kucing** tidak akan berubah menjadi objek **Mamalia**.
 - Objek tetap berupa **Kucing**, namun *hanya bisa diperlakukan* layaknya **Mamalia** lainnya dan *sifat* Kucing-nya “disembunyikan” sampai di-*downcast* ke **Kucing** kembali.
- Casting dapat *memperluas* ataupun *mempersempit* hal-hal yang bisa dilakukan atas suatu **objek**.

JENIS CASTING

- **Upcasting** (widening) → Casting dari suatu **subclass** menjadi **superclass**.
 - Lebih fleksibel untuk mengakses *class member* dari **superclass** akan tetapi tidak dapat mengakses *class member* milik **subclass**.
- **Downcasting** (narrowing) → Casting dari suatu **superclass** menjadi **subclass**.

UPCASTING

- Ketika variabel referensi dari **superclass** merujuk ke objek **subclass**, ini dikenal sebagai upcasting. Dapat dilakukan secara *eksplisit* maupun *implisit*.

```
public class Children extends Parent {  
    int id;  
  
    @Override  
    void method(){  
        System.out.println(x:"Method from child");  
    }  
  
    public void childrenMethod(){  
        System.out.println(x:"Only for children");  
    }  
}
```

```
public class Parent {  
    String name;  
  
    void method(){  
        System.out.println(x:"Method from parent");  
    }  
}
```

```
public class CastingObjek {  
    Run | Debug  
    public static void main(String[] args) {  
        //Upcasting  
        Parent p = new Children(); //implicit upcasting  
        p.name = "Joko";  
  
        System.out.println(p.name);  
        p.method();  
    }  
}
```

```
Joko  
Method from child  
PS D:\CODING JAVA\00 - Kuliah FPA\P13\P13>
```

UPCASTING

- Ketika variabel referensi dari **superclass** merujuk ke objek **subclass**, ini dikenal sebagai upcasting. Dapat dilakukan secara *eksplisit* maupun *implisit*.

```
public class Children extends Parent {  
    int id;  
  
    @Override  
    void method(){  
        System.out.println(x:"Method from child");  
    }  
  
    public void childrenMethod(){  
        System.out.println(x:"Only for children");  
    }  
}
```

```
public class Parent {  
    String name;  
  
    void method(){  
        System.out.println(x:"Method from parent");  
    }  
}
```

```
public class CastingObjek {  
    Run | Debug  
    public static void main(String[] args) {  
        //Upcasting  
        Children c = new Children();  
        Parent p = (Parent) c; //explicit upcasting  
        p.name = "Joko";  
  
        System.out.println(p.name);  
        p.method();  
    }  
}
```

```
Joko  
Method from child  
PS D:\CODING JAVA\00 - Kuliah FPA\P13\P13>
```

MENGAPA UPCASTING?

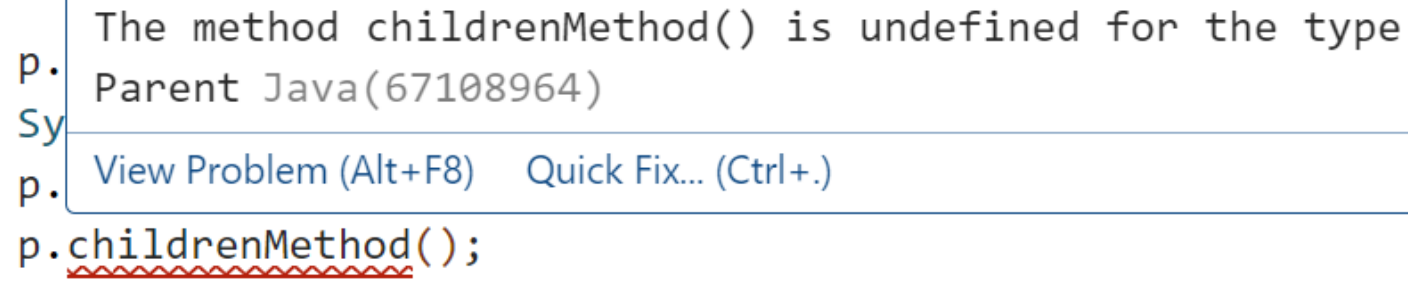
- Kita perlu *upcasting* ketika kita ingin menulis kode secara umum yang **HANYA** berhubungan dengan tipe superclass-nya
- Contoh:
 - Dengan *upcasting*, kita telah **membatasi** jumlah *method* yang tersedia bagi objek **Children**, tetapi tidak mengubah objek itu sendiri.

```
Parent p = new Children();
```

- Sekarang, kita **tidak bisa** melakukan apapun yang khusus untuk **Children**. Misalnya, kita tidak bisa memanggil method `childrenMethod()` dari variabel referensi bertipe **Parent**.

UPCASTING

- Meskipun objek **Parent** tetap berupa objek **Parent** (tidak berubah objek) dan merujuk ke objek **Kucing** maka *reference variable* **p** ketika mencoba memanggil method **childrenMethod()** akan mengakibatkan *compiler error*.



The screenshot shows an IDE window with a code editor and a tooltip. The code editor contains the following text:
p.
Sy
p.
p.childrenMethod();
The tooltip, which appears to be a hover over the underlined method call, contains the following text:
The method childrenMethod() is undefined for the type
Parent Java(67108964)
View Problem (Alt+F8) Quick Fix... (Ctrl+.)

- Untuk dapat memanggil method **childrenMethod()** , kita perlu melakukan *downcasting*

DOWNCASTING

- Dengan *downcasting*, kita bisa menggunakan variabel referensi bertipe **Parent** untuk memanggil *method* yang hanya tersedia untuk kelas **Children**.
- *Downcasting* = casting dari **superclass** ke **subclass**.
- Untuk memanggil *method* **childrenMethod()**, kita perlu *downcasting* **Parent** ke **Children**.

```
p.name = "Joko";  
System.out.println(p.name);  
p.method();  
((Children)p).childrenMethod(); //downcasting
```

variabel referensi **p** di-*downcasting* ke **Children** agar bisa memanggil *method* **childrenMethod()**

```
Joko  
Method from child  
Only for children  
PS D:\CODING JAVA\00 - Kuliah FPA\P13\P13>
```

INSTANCEOF

- Tidak seperti *upcasting*, proses *downcasting* bisa **gagal** jika tipe objek sebenarnya **bukan** tipe objek target *downcasting*.

```
public class TestInstance {  
    Run | Debug  
    public static void main(String[] args) {  
        Parent p = new Father();  
        ((Children)p).childrenMethod();  
    }  
}
```

```
Exception in thread "main" java.lang.ClassCastException: casting.Father cannot  
    be cast to casting.Children  
        at casting.TestInstance.main(TestInstance.java:6)  
PS D:\CODING JAVA\00 - Kuliah FPA\P13\P13>
```

INSTANCEOF

- Tidak seperti *upcasting*, proses *downcasting* bisa **gagal** jika tipe objek sebenarnya **bukan** tipe objek target *downcasting*.
- Untuk mengatasinya, kita perlu operator **instanceof** untuk memeriksa tipe suatu objek sebelum melakukan *downcasting*.

```
public class TestInstance {  
    Run | Debug  
    public static void main(String[] args) {  
        Parent p = new Father();  
        if(p instanceof Children){  
            /*  
             * Jika p merujuk ke objek Children maka proses -  
             * downcasting akan dilakukan  
             */  
            ((Children)p).childrenMethod();  
        }  
    }  
}
```

QUICK QUIZ – OUTPUT?

```
class Binatang {  
    public String bermain(){  
        return "Binatang bermain";  
    }  
}  
class Kucing extends Binatang{  
    public String bermain(){  
        return "Bermain bola";  
    }  
}  
class Hamster extends Binatang{  
    public String bermain(){  
        return "Bermain dalam kincir putar";  
    }  
}
```

```
public class JavaGenericTest{  
    public static void main(String[] args) {  
        Binatang binatang = new Binatang();  
        Hamster hamtarro = new Hamster();  
        Binatang oyen = new Kucing();  
        binatang = hamtarro;  
  
        System.out.println(binatang.bermain());  
        System.out.println(oyen.bermain());  
        System.out.println(hamtarro.bermain());  
    }  
}
```

TERIMA KASIH