Dasar Dasar Pemrograman 2

Acuan: Introduction to Java Programming and Data Structure, Bab 11

Sumber Slide: Liang,

Dimodifikasi untuk Fasilkom UI oleh Ade Azurat

Topik: Polymorphism

Ingat contoh program Cards pekan lalu?

https://greenteapress.com/thinkjava7/html/chapter-14.html

- Pada class CardCollection.java:

```
public void deal(CardCollection that, int n) {
   for (int i = 0; i < n; i++) {
      Card card = this.popCard();
      that.addCard(card);
   }
}
```

- Pada class Eights.java:

```
discardPile = new Hand("Discards");
deck.deal(discardPile, 1);
```

Ingat contoh program Cards pekan lalu?

https://greenteapress.com/thinkjava7/html/chapter-14.html

- Pada class CardCollection.java:

```
public void deal(CardCollection that , int n) {
   for (int i = 0; i < n; i++) {
      Card card = this.popCard();
      that.addCard(card);
   }
}
```

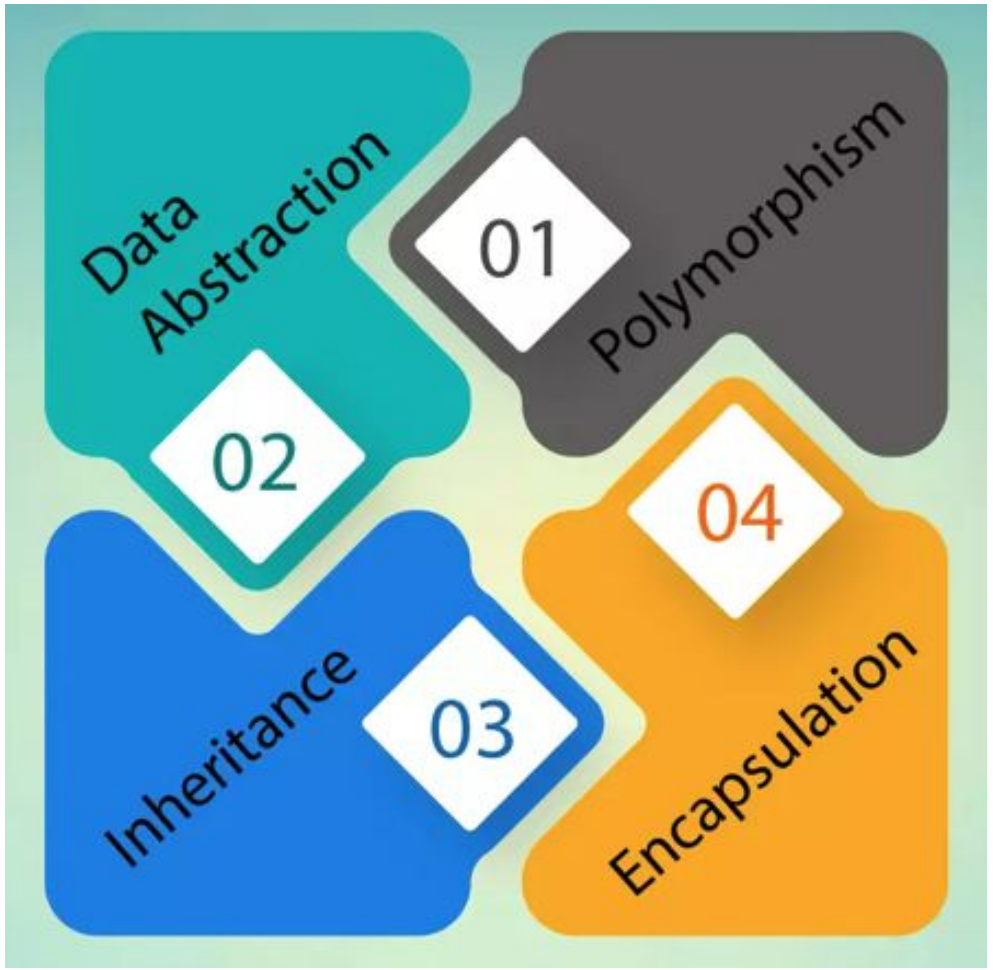
Pada deklarasi ***method deal***, parameter-nya adalah class ***CardCollection*** dan int.
Namun pada pemanggilan di class Eight, dipanggill dengan object yang dideklarasikan bertipe ***Hand***?

- Pada class Eights.java:

```
discardPile = new Hand ("Discards");
deck.deal(discardPile,  1);
```

Review

Data Abstraction: Abstract / hide unnecessary information



Polymorphism:
??? Pelajaran kita hari ini! ☺️

Inheritance: Inherits properties/field or methods from other (parent) class.

Encapsulation: Encapsulate many data into one single class

Sumber gambar: https://images.theengineeringprojects.com/image/webp/2021/11/6-2.jpg.webp?ssl=1

Objectives

- To define a subclass from a superclass through inheritance (§11.2).
- To invoke the superclass's constructors and methods using the **super** keyword (§11.3).
- To override instance methods in the subclass (§11.4).
- To distinguish differences between overriding and overloading (§11.5).
- To explore the **toString()** method in the **Object** class (§11.6).
- To discover polymorphism and dynamic binding (§§11.7–11.8).
- To describe casting and explain why explicit downcasting is necessary (§11.9).
- To explore the **equals** method in the **Object** class (§11.10).
- To store, retrieve, and manipulate objects in an **ArrayList** (§11.11).
- To implement a **Stack** class using **ArrayList** (§11.12).
- To enable data and methods in a superclass accessible from subclasses using the **protected** visibility modifier (§11.13).
- To prevent class extending and method overriding using the **final** modifier (§11.14).

Inheritance and Polymorphism

- A class defines a type.
- A type defined by a subclass is called a *subtype*, and a type defined by its superclass is called a *supertype*.
- Therefore, you can say that **Circle** is a subtype of **GeometricObject** and **GeometricObject** is a supertype for **Circle**.

Polymorphism means that a variable of a supertype can refer to a subtype object.

In other words: Polymorphism means many forms!

Polymorphism and Dynamic Binding

```
public class PolymorphismDemo {
    public static void main(String[] args) {
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }

    public static void m(Object x) {
        System.out.println(x.toString());
    }
}

class GraduateStudent extends Student {
}

class Student extends Person {
    public String toString() {
        return "Student";
    }
}

class Person extends Object {
    public String toString() {
        return "Person";
    }
}
```

Method ***m*** takes a parameter of the Object type. You can invoke it with any object.

An object of a subtype can be used wherever its supertype value is required. This feature is known as ***polymorphism***.

- When the method ***m(Object x)*** is executed, the argument ***x***'s toString method is invoked. ***x*** may be an instance of ***GraduateStudent***, ***Student***, ***Person***, or ***Object***.
- Classes ***GraduateStudent***, ***Student***, ***Person***, and ***Object*** have their own implementation of the ***toString*** method.
- Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime.
- This capability is known as ***dynamic binding***.

Polymorphism and binding

- ***Polymorphism*** allows an object to take multiple forms – when a method exhibits polymorphism, the compiler has to map the name of the method to the final implementation.
- If it's mapped at **compile time, it's a static or early binding.**
  - Such as, static, private method.
- If it's resolved at **runtime, it's known as dynamic or late binding.**
  - Object method.

Dynamic Binding

Dynamic binding works as follows:

- Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n.
- That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the Object class.
- If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found.
- Once an implementation is found, the search stops and the first-found implementation is invoked.

Method Matching vs. Binding

- Matching a method signature and binding a method implementation are two issues.
- The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time.
- A method may be implemented in several subclasses.
- The Java Virtual Machine dynamically binds the implementation of the method at runtime.

Polymorphism Demo

- Polymorphism allows methods to be used generically for a wide range of object arguments.
- If a method's parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter's subclasses (e.g., Student or String).
- When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically.

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }
}

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Ayo berhenti dulu!

- Apa itu polymorphism?
- Apa keterkaitannya dengan dynamic binding?

Casting Objects (PolymorphismDemo.java)

You have already used the casting operator to convert variables of one primitive type to another. *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```
m(new Student());
```

assigns the object new Student() to a parameter of the Object type. This statement is equivalent to:

```
Object o = new Student();  // Implicit casting - upcasting
m(o);
```

The statement Object o = new Student(), known as implicit casting, is legal because an instance of Student is automatically an instance of Object.

Casting

Su
St

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Type mismatch: cannot convert from Object to Student

        at ErrorCasting.main(ErrorCasting.java:4)
```

```
Object o = new Student();
Student b = o;
```

A compile error would occur.

Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't?

This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student.

Even though you can see that o is really a Student object, the compiler is not so clever to know it.

Explicit Casting – Down casting

Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

```
Object o  = new Student();
Student b = o;
```

- To tell the compiler that o is a Student object, use an explicit casting.
- The syntax is similar to the one used for casting among primitive data types.
- Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student) o; // Explicit casting – down casting
```

Downcasting:
Casting from *Superclass* to *Subclass*

Explicit casting (down-casting) must be used when casting an object from a superclass to a subclass.

This type of casting may not always succeed.

```
Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

The `instanceof` Operator

Use the `instanceof` operator to test whether an object is an instance of a class:

```
Object o = new GraduateStudent(); // implicit casting (upcasting)
if (o instanceof GraduateStudent) {
   System.out.println("Object o is an instance of GraduateStudent.");
   GraduateStudent gs = (GraduateStudent) o; // explicit casting
}
```

Casting

- ✔ To help understand casting, you may also consider the analogy of fruit, apple, and orange with the Fruit class as the superclass for Apple and Orange.
- ✔ An apple is a fruit, so you can always safely assign an instance of Apple to a variable for Fruit.
- ✔ However, a fruit is not necessarily an apple, so you have to use explicit casting to assign an instance of Fruit to a variable of Apple.

UNIVERSITAS INDONESIA

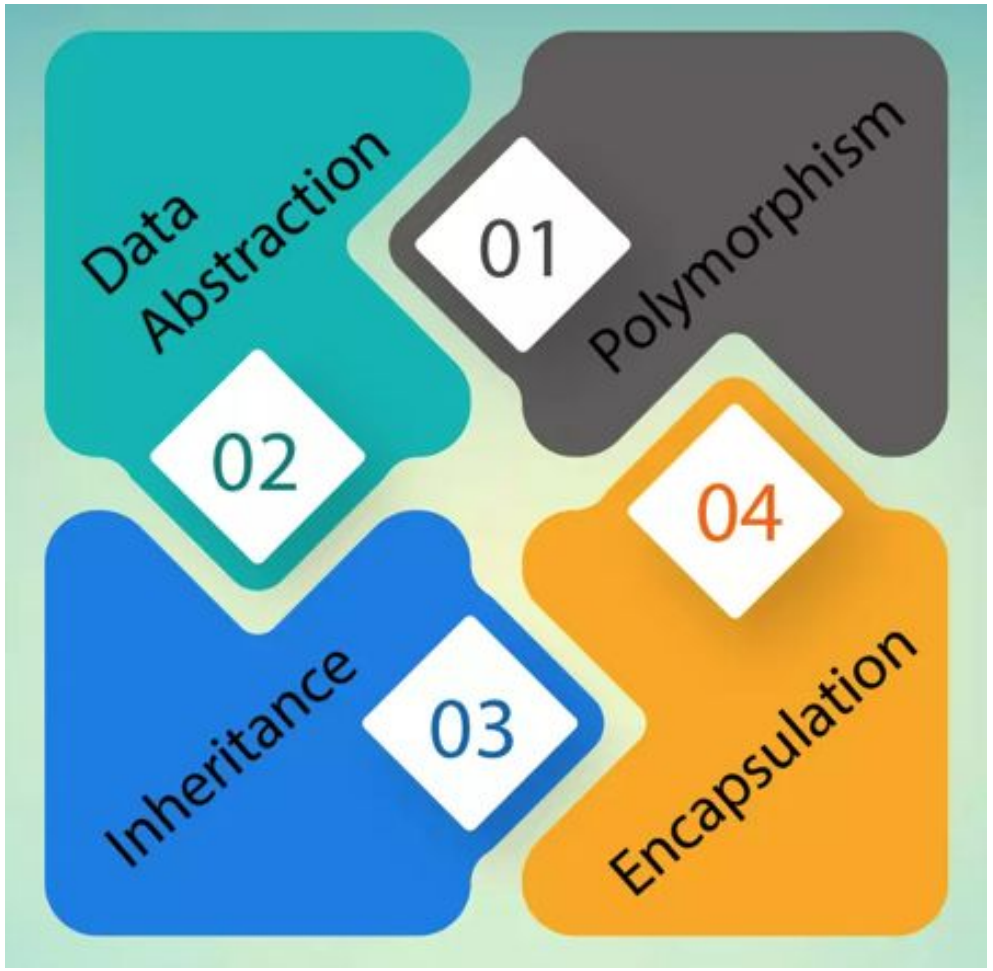
FAKULTAS ILMU KOMPUTER

Ayo berhenti dulu!

- ❑ Apa itu casting?
- ❑ Apakah casting bisa mengakibatkan runtime error?
- ❑ Kapan harus melakukan implicit vs explicit casting?

Pillar of Object Oriented Programming

Data Abstraction: Abstract / hide unnecessary information



Polymorphism:
Many form, an object can be others sub-class.

Inheritance: Inherits properties/field or methods from other (parent) class.

Encapsulation: Encapsulate many data into one single class

Sumber gambar: https://images.theengineeringprojects.com/image/webp/2021/11/6-2.jpg.webp?ssl=1

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Application of Polymorphism

Some basic consequences and brief introduction into Generics

The `equals` Method

The `equals()` method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

```
public boolean equals(Object obj) {
  return this == obj;
}
```

For example, the equals method is overridden in the Circle class.

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

NOTE

- ✔ The == comparison operator is used for comparing **two primitive data type values** or for determining whether ***two objects have the same references***.
- ✔ The ***equals*** method is intended to test whether ***two objects have the same contents***, provided that the method is modified in the defining class of the objects.
- ✔ The == operator is stronger than the ***equals*** method, in that the == operator checks whether the ***two reference*** variables refer to the same object.

The ArrayList Class

You can create an array to store objects. But the array's size is fixed once the array is created. Java provides the ArrayList class that can be used to store an unlimited number of objects.

java.util.ArrayList<E>	
+ArrayList()	Creates an empty list
+add(o: E) : void	Appends a new element o at the end of this list.
+add(index: int, o: E) : void	Adds a new element o at the specified index in this list.
+clear(): void	Removes all the elements from this list.
+contains(o: Object): boolean	Returns true if this list contains the element o.
+get(index: int) : E	Returns the element from this list at the specified index.
+indexOf(o: Object) : int	Returns the index of the first matching element in this list.
+isEmpty(): boolean	Returns true if this list contains no elements.
+lastIndexOf(o: Object) : int	Returns the index of the last matching element in this list.
+remove(o: Object): boolean	Removes the element o from this list.
+size(): int	Returns the number of elements in this list.
+remove(index: int) : boolean	Removes the element at the specified index.
+set(index: int, o: E) : E	Sets the element at the specified index.

Generic Type

- ***ArrayList<E>*** is known as a generic class with a generic type *E*.
- You can specify a concrete type to replace E when creating an ArrayList.
- For example, the following statement creates an ArrayList and assigns its reference to variable cities.
- This ArrayList object can be used to store strings.

```
ArrayList<String> cities = new ArrayList<String>();
```

```
ArrayList<String> cities = new ArrayList<>();
```

Differences and Similarities between Arrays and ArrayList

Operation	*Array*	*ArrayList*
Creating an array/ArrayList	`String[] a = new String[10]`	`ArrayList<String> list = new ArrayList<>();`
Accessing an element	`a[index]`	`list.get(index);`
Updating an element	`a[index] = "London";`	`list.set(index, "London");`
Returning size	`a.length`	`list.size();`
Adding a new element		`list.add("London");`
Inserting a new element		`list.add(index, "London");`
Removing an element		`list.remove(index);`
Removing an element		`list.remove(Object);`
Removing all elements		`list.clear();`

Array Lists from/to Arrays

Creating an ArrayList from an array of objects:

```
String[] array = {"red", "green", "blue" };
ArrayList<String> list = new ArrayList<>(Arrays.asList(array));
```

Creating an array of objects from an ArrayList:

```
String[] array1 = new String[list.size()];
list.toArray(array1);
```

max and min in an Array List

```
String[] array = {"red", "green", "blue" };
System.out.println(java.util.Collections.max(
    new ArrayList<String>(Arrays.asList(array))));
```

```
String[] array = {"red", "green", "blue" };
System.out.println(java.util.Collections.min(
    new ArrayList<String>(Arrays.asList(array))));
```

Shuffling an Array List

```
Integer[] array = {3, 5, 95, 4, 15, 34, 3, 6, 5};
ArrayList<Integer> list = new ArrayList<>(Arrays.asList(array));
java.util.Collections.shuffle(list);
System.out.println(list);
```

The MyStack Classes

A stack to hold objects.

MyStack	
-list: ArrayList	A list to store elements.
+isEmpty(): boolean	Returns true if this stack is empty.
+getSize(): int	Returns the number of elements in this stack.
+peek(): Object	Returns the top element in this stack.
+pop(): Object	Returns and removes the top element in this stack.
+push(o: Object): void	Adds a new element to the top of this stack.
+search(o: Object): int	Returns the position of the first element in the stack from the top that matches the specified element.

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Ayo berhenti dulu!

- ❑ Masih ingat kapan kamu terakhir menggunakan Array? Kalau ArrayList?
- ❑ Apa saja perbedaan antara Array dengan ArrayList?

The `protected` Modifier

- The `protected` modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.

- private, default, protected, public


```
private, none (if no modifier is used), protected, public
```

Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Visibility Modifiers

package p1;

```
public class C1 {
  public int x;
  protected int y;
  int z;
  private int u;

  protected void m() {
  }
}
```

```
public class C2 {
  C1 o = new C1();
  can access o.x;
  can access o.y;
  can access o.z;
  cannot access o.u;

  can invoke o.m();
}
```

```
public class C3
        extends C1 {
  can access x;
  can access y;
  can access z;
  cannot access u;

  can invoke m();
}
```

package p2;

```
public class C4
        extends C1 {
  can access x;
  can access y;
  cannot access z;
  cannot access u;

  can invoke m();
}
```

```
public class C5 {
  C1 o = new C1();
  can access o.x;
  cannot access o.y;
  cannot access o.z;
  cannot access o.u;

  cannot invoke o.m();
}
```

A Subclass Cannot Weaken the Accessibility

- A subclass may override a protected method in its superclass and change its visibility to public.
- However, a subclass cannot weaken the accessibility of a method defined in the superclass.
- For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

NOTE

- The modifiers are used on classes and class members (data and methods),
- except that the final modifier can also be used on local variables in a method.
- A final local variable is a constant inside a method.

The `final` Modifier

- The `final` class cannot be extended:

```
final class Math {
  ...
}
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.

UNIVERSITAS INDONESIA

FAKULTAS ILMU KOMPUTER

Ayo berhenti dulu!

- ❑ Coba diingat lagi modifier public, default, dan private yang sudah pernah kamu gunakan!
- ❑ Kapan kamu harus menggunakan modifier protected? final?

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Selamat Berlatih!

Perhatikan lagi List Objective yang perlu dikuasai pekan ini.
Baca buku acuan dan berlatih!
Bila masih belum yakin tanyakan ke dosen, tutor atau Kak Burhan.

Semangat !