



Chapter 2: Java OO II



XIANG ZHANG

javacose@qq.com

Content

2

- Abstraction
 - Abstract Class
 - Interface
- Inheritance
- Polymorphism

Abstraction



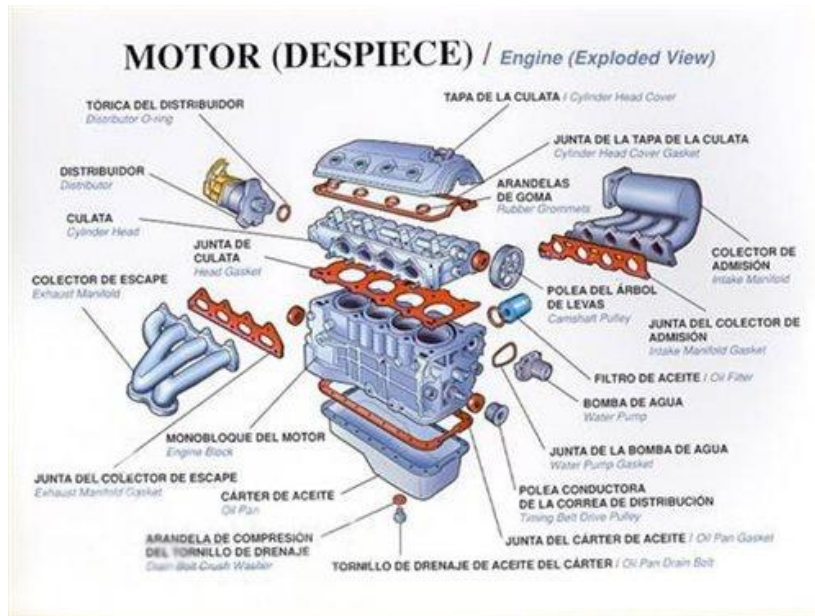
Abstraction

4

- What is **Abstraction**?
 - “An abstraction is a general idea rather than one relating to a particular object, person, or situation.” - From Collins
- The Significance of Abstraction
 - Model
 - Implementation
- Language Tools for Abstraction in Java
 - Abstract Class
 - Interface

Abstraction

5



Abstraction – Abstract Class

6

- Abstract Class : Java Class providing part of implementations
 - Including abstract methods
 - Cannot be used to create objects
 - Must be inherited and implemented

Abstraction – Abstract Class



An Abstract Class for Benchmark

```
public abstract class Benchmark {  
  
    private final int COUNT = 100;  
  
    abstract void benchmark();  
  
    public final long repeat() {  
        long start = System.nanoTime();  
        for (int i = 0; i < COUNT; i++) {  
            benchmark();  
        }  
        return ((System.nanoTime() - start)) / COUNT;  
    }  
}
```

Judger



Abstraction – Implemented Class



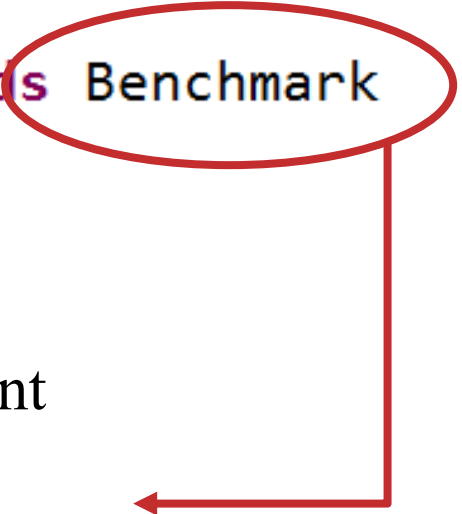
An Implemented Class for Benchmark

```
public class BenchmarkImpl extends Benchmark {  
  
    @Override  
    void benchmark() {  
        // do some algorithm  
    }  
  
    public static void main(String[] args) {  
        long time = new BenchmarkImpl().repeat();  
        System.out.println(time);  
    }  
}
```

Player




```
public class BenchmarkImpl extends Benchmark
```



A contract:

1. All its subclasses must implement `benchmark()` method;
2. No overriding on `repeat()` method is allowed, preventing cheating on timekeeping.

Lab Work

10

- A benchmark for sorting algorithms
- Input: an Integer array to be sorted
 - `Integer[] input = new Integer[1000]`
- Functions of abstract class
 - randomly generating an input array;
 - measuring the time efficiency of sorting;
 - judging the correctness of sorting;
 - comparing your time efficiency to the efficiency of java sorting algorithm

Lab Work

11



```
import java.util.Random;
```



```
import java.util.Arrays;
```

Abstraction – Interface

12

- The language basis of Java is Class
- But the basis of OOD is Type
- Class \cong Type + Implementation
- Java prompts Interface-based OOD
 - Designer only cares of design, or saying, interface;
 - Developer cares of implementation, or saying, classes.

Abstraction – Interface

13

- Most Interfaces do:
 - Distinguish one type with other types
 - Showing that one type can have the **ability** to do sth:
 - ✦ Cloneable : an object of this type can be *cloned*
 - ✦ Comparable : objects of this type can be *compared*
 - ✦ Runnable : an object of this type can be *run* in a thread
 - ✦ Serializable : an object of this type can be *serialized*

Example: Tweetable

14

```
public interface Tweetable {  
    public void tweet();  
}
```

```
public class Sparrow implements Tweetable {
```

```
    @Override  
    public void tweet() {  
        System.out.println("JiuJiu~");  
    }  
}
```

```
}
```

Tweetable



```
public class Hummingbird {
```

```
    public void fly(Place a, Place b){  
    }  
}
```

```
}
```

Non-tweetable



Abstraction – Interface

16

- Defining an Interface
 - Constant
 - Method
 - Nested Class and Interface

```
public interface Comparable<T> {  
    int compareTo(T obj);  
}
```


Abstraction - Interface

17

- Interface
 - Is a contract between designer and programmer;
 - Programmer must fulfill the interface with the **appearance** of a type;
 - Designer doesn't care about anything of the inner implementation

```
public class Point implements Comparable<Point> {
    private int x, y;
    private static final Point ORIGIN = new Point(0, 0);

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public double distance(Point p) {
        int xdiff = x - p.x;
        int ydiff = y - p.y;
        return Math.sqrt(xdiff * xdiff + ydiff * ydiff);
    }

    public int compareTo(Point p) {
        double pDist = p.distance(ORIGIN);
        double dist = this.distance(ORIGIN);
        if (dist > pDist)
            return 1;
        else if (dist == pDist)
            return 0;
        else
            return -1;
    }
}
```

Abstraction – Interface

19

- Fields in Interface

- public
- static
- Final
- Must be initialized

```
public interface Color {  
    int white = 0; // public static final  
    int black = 1; // public static final  
}  
...  
public class ColorImpl implements Color{  
    public static void main(String[] args){  
        Color c = new ColorImpl();  
        c.black = 2; // Compiling error!  
    }  
}
```

Abstraction – Interface

20

- Implementing an Interface
 - A Class can implement multiple interfaces
 - Multiple Implementation
 - All methods in an interface should be implemented
 - Multiple Implementation != Multiple Inheritance

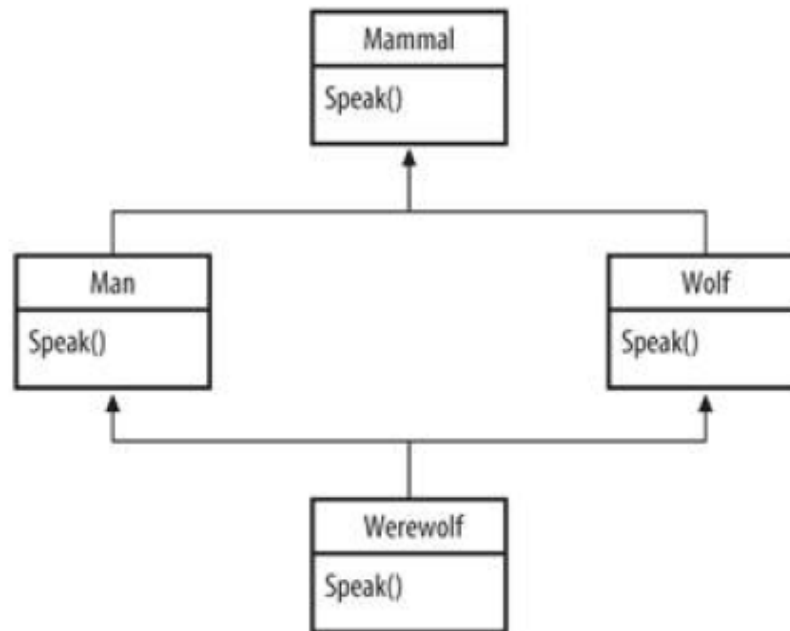
```
class Point implements Comparable<Point>, Serializable, Cloneable{  
    ...  
}
```

Abstraction – Interface

21

- Think

- Does M-Implementation bring the same problem as M-Inheritance?



Abstraction – Interface

22

- Using Interface to declare the type of objects

```
Comparable<Point> obj = new Point();
```

```
double distance = obj.distance(p1);
```

```
// INVALID: Comparable has no distance method
```

```
double distance = (Point)obj.distance(p1);
```

```
//OK!
```

```
String obj_string = obj.toString();
```

```
//OK!
```

Abstraction – Interface

23

- Interface-based Programming

- Agile: First, let's care about the **design**, the **type**, the **interface**, and ignore the **implementation**, the **class**.
- Reliable: assure the correctness based on the use of **types**.

```
public class Sorter {  
    public static Comparable<?>[] sort(Comparable<?>[] list) {  
        // implementation details ...  
        return list;  
    }  
}
```

Abstraction – Interface

24

- **Marker Interface**
 - Nothing defined in an interface
 - It is just a marker
 - Such as Cloneable

Abstraction – Interface

25

- Abstract Class vs. Interface
 - **Multiple inheritance** is allowed for Interface, not for abstract class.
 - Abstract class provides part of **implementation**, while interface has no implementation.

Abstract – Interface

26

- Combining Abstract Class and Interface

```
class ThreeDPoint extends Point implements Comparable<ThreeDPoint>{  
    ...  
}
```

Inheritance



Inheritance

28

- The Significance of Inheritance
 - Code **reuse**
 - Enhancement of **maintainability**
 - Enhancement of **scalability**
- Types of Inheritance
 - Inheritance of class
 - Inheritance of interface

Inheritance – Class

29

- Example - House

```
public class House {  
    private String doorStyle;  
    private String windowsStyle;  
    private String wallStyle;  
  
    public House(String door, String windows, String wall){  
        doorStyle = door;  
        windowsStyle = windows;  
        wallStyle = wall;  
    }  
  
    // Getters and Setters goes here...  
}
```

Inheritance – Class

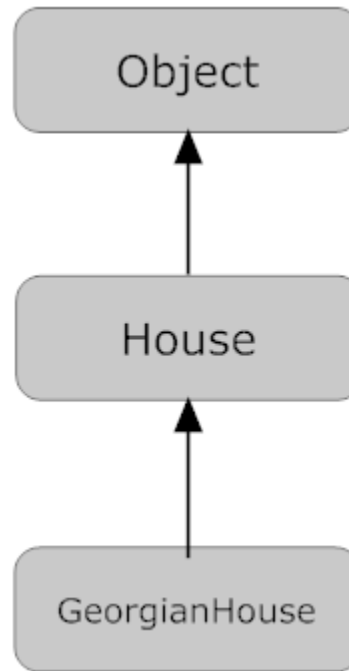
30

- Example - GeorgianHouse

```
public class GeorgianHouse extends House{  
    private String EavesStyle;  
    public GeorgianHouse(String door, String windows, String wall, String eaves){  
        super(door, windows, wall);  
        EavesStyle = eaves;  
    }  
    // Getters and Setters goes here...  
}
```

Inheritance – Class

31



Inheritance – Class

32

- Constructors in Inheritance

1. Constructors in subclasses should invoke constructors in superclass **explicitly** for initialization.
2. If step 1 is not satisfied, the **default** constructor in superclass will be invoked.
3. If no default constructor, but a non-default constructor is defined in superclass, step 1 **must be** satisfied.
4. The invocation of constructors in superclass should be placed **foremost**.

Inheritance - Class

33

- Thinking in Java
 - A **robust** programming language should have a sound initialization process.
 - Each field in a class should be **initialized**.

Inheritance – Class

34

- Overriding (覆盖)
 - Overriding NOT Overloading (重载) //what is overloading
 - Overriding means a method in subclass will replace a method with same **signature** in superclass //what is signature
 - Overriding let the subclass perform differently with superclass

Inheritance – Class

35

- Example

```
public class House{  
    protected void doorOpen(){  
        System.out.println  
            ("Door opened inward");  
    }  
}  
  
public class GeorgianHouse(){  
    protected void doorOpen(){  
        System.out.println  
            ("Door opened outward");  
    }  
}
```

Inheritance – Want to invoke super method?

36

```
public class Father {  
  
    public void test(){  
        System.out.println("father");  
    }  
}
```

```
public class Son extends Father{  
  
    public void test(){  
        System.out.println("son");  
    }  
}
```

```
public static void main(String[] args){  
    Son s = new Son();  
    ((Father)s).test();  
}
```

- Is it possible to invoke super method by casting? **NO**
- The only way is to use **super.test();** in Son

Inheritance – Class

37

- Limits of Overriding

- The access rights should be **enlarged** or **unchanged**, not be **reduced**. // why?
- The return type should be **reduced** or **unchanged**, not be **enlarged**. // why?
 - ✦ If return type in superclass is Class A, the return type in subclass should $\subseteq A$
 - ✦ If return type in superclass is a primary type, it should be unchanged in subclass.

Inheritance -Class

38

- Thinking in Java
 - Overriding MUST retain compatibility (not breaking the behavior specification in superclass)
 - Think
 - ✦ `House spark_house = new GeorgianHouse();` // Right?
 - ✦ `GeorgianHouse spark_house = new House();` //Right?

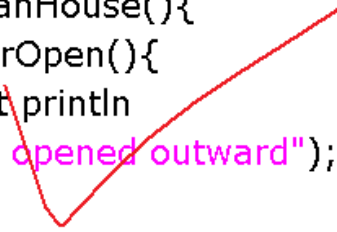
Inheritance – Class

39

- Examples

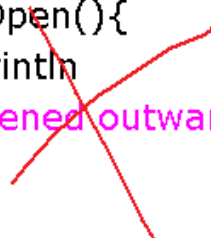
```
public class House{  
    protected void doorOpen(){  
        System.out.println  
            ("Door opened inward");  
    }  
}
```

```
public class GeorgianHouse(){  
    public void doorOpen(){  
        System.out.println  
            ("Door opened outward");  
    }  
}
```



```
public class House{  
    protected void doorOpen(){  
        System.out.println  
            ("Door opened inward");  
    }  
}
```

```
public class GeorgianHouse(){  
    private void doorOpen(){  
        System.out.println  
            ("Door opened outward");  
    }  
}
```



Inheritance – Class

40

- Keyword: super

```
public class House{
    protected void doorOpen(){
        System.out.println
            ("Door opened inward");
    }
}

public class GeorgianHouse(){
    protected void doorOpen(){
        super.doorOpen();
        System.out.println
            ("Door opened outward");
    }
}
```


Inheritance – Class

41

- Hiding

```
public class House{
    public String className = "House";
    public void showName(){
        System.out.println
            ("The super class: " + className);
    }
}

public class GeorgianHouse(){
    public String className = "GeorgianHouse";
    public void showName(){
        System.out.println
            ("The extended class: " + className);
    }
}
```

Guess

42

A:

GeorgianHouse

House

The extended class: GeorgianHouse

The extended class: GeorgianHouse

B:

GeorgianHouse

GeorgianHouse

The extended class: GeorgianHouse

The extended class: GeorgianHouse

C:

GeorgianHouse

House

The extended class: GeorgianHouse

The super class: House

```
public static void main(String[] args){  
    GeorgianHouse ghouse = new GeorgianHouse();  
    House house = ghouse;  
    System.out.println(ghouse.className);  
    System.out.println(house.className);  
    ghouse.showName();  
    house.showName();  
}
```

D:

GeorgianHouse

House

The extended class: GeorgianHouse

The extended class: House



for Field, look at the declaration type;
for Method, look at the run-time type;

declared type: House
runtime type: GeorgianHouse

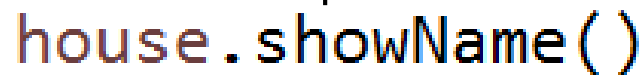
The value of a field depends
on the declared type

`house.className`



The existence of a field/method depends on the declared type

`house.showName()`



The result of a method
depends on the runtime type

Inheritance – Class

44

- Conversion of Types
 - Objective : convert an object of one class to another
 - For example: a Parrot to a Bird, gHouse to House
- Classification of Conversion
 - Upcasting 上溯造型
 - Downcasting 下溯造型

Inheritance – Class

45

- Upcasting

```
GeorgianHouse ghouse = new GeorgianHouse();  
((House)ghouse).showname();
```

- Upcasting is safe and unrestricted
- Downcasting :
 - reverse of Upcasting,
 - may be unsafe
 - may cause casting exceptions
- Use **instanceOf** to check the safety

Self-study

46

- RTTI : Run-Time Type Identification
- Understand RTTI can help you understand
 - Type conversion in Java
 - Polymorphism
 - Reflection in J2EE
- Read : Think in Java Chapter 10

Inheritance – Class

47

- Keyword: final
 - **final** before a class means it is **not inheritable**
 - **final** before a method means it is **not overridable**

```
final class GeorgianHouse{  
    ...  
}
```

```
public class SomeHouse extends GeorgianHouse{  
    ...  
}
```

```
public class GeorgianHouse{  
    protected final void showName(){  
        ...  
    }  
}
```

```
public class SomeHouse extends GeorgianHouse{  
    protected void showName(){  
        ...  
    }  
}
```

Self-study

48

- Class Inheritance : How and when?
 - Is-a (Inheritance)
 - Has-a (Composition)
- How to design an extensible class
- Read:
 - Java Programming Language 3.11和3.12
 - Thinking in Java chapter 6

Inheritance – Interface

49

- Multiple Inheritance for Interfaces

```
public interface SerializableRunnable
    extends java.io.Serializable, Runnable
{
    ...
}
```

- Inheritance of Interfaces also has:

- Hiding of fields
- Override of Methods

Polymorphism



Polymorphism

51

- Definition of Polymorphism
 - Greek, means “Multiple Forms”
 - Refers to the existence of different methods with same names
 - Intuition: in OOD, same objects have the same behavior (method name), but have different way of behaving
 - Question: What will the exact behaving result be for a certain object?
 - Significance: Improve the flexibility and versatility in OOP
- Types of Polymorphism
 - Static polymorphism - polymorphism in compile-time
 - Dynamic polymorphism - polymorphism in run-time

Polymorphism

52

- Static Polymorphism

- Polymorphism which can be determined in compilation
- Overloading

`Calculator.add(10, 9)`

`Calculator.add(0.5, 0.4)`

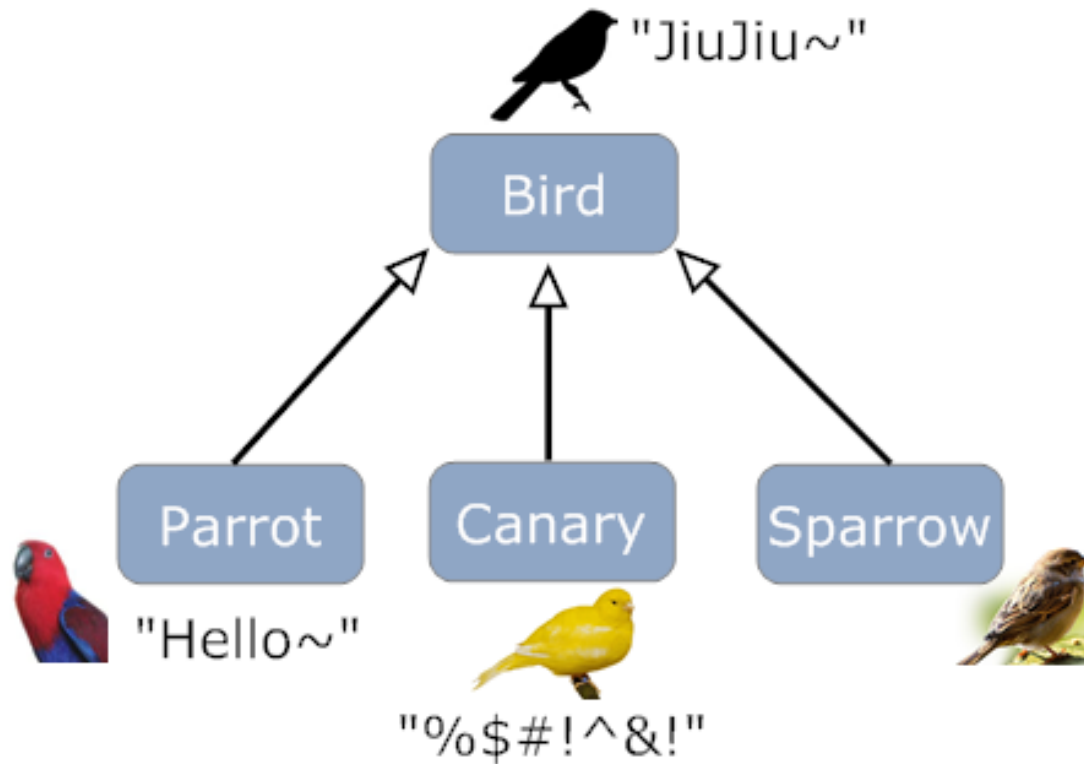
Polymorphism

53

- Dynamic Polymorphism
 - Behavior and result be can only determined in run-time
 - Dynamic Binding – The binding of method invocation and method body in run-time
 - Classification of Dynamic Polymorphism
 - ✦ Inheritance Polymorphism
 - ✦ Interface Polymorphism

Polymorphism

54



Polymorphism

55

- Inheritance Polymorphism

```
Bird p; // 声明但不创建对象
p = new Parrot(); //创建对象并引用
p.tweet(); // "Hello~"
p = new Canary(); //运行期动态改变对象引用
p.tweet(); //对象的行为发生变化
p = new Sparrow();
p.tweet();
```

Polymorphism

56

- Interface Polymorphism

```
public interface Tweetable{  
    public void tweet();  
}
```

```
public class Parrot implements Tweetable{  
    public void tweet(){ ... }  
}
```

```
public class Canary implements Tweetable{  
    public void tweet(){ ... }  
}
```

```
public class Sparrow implements Tweetable{  
    public void tweet(){ ... }  
}
```

```
Tweetable t;  
t = new Parrot();  
t.tweet();  
t = new Canary();  
t.tweet();  
...
```


Something Strange

57

```
public class House{  
    public String className = "House";  
    public void showName(){  
        System.out.println  
            ("The super class: " + className);  
    }  
}
```

```
public class GeorgianHouse(){  
    public String className = "GeorgianHouse";  
    public void showName(){  
        System.out.println  
            ("The extended class: " + className);  
    }  
}
```

```
public static void main(String[] args){  
    GeorgianHouse ghouse = new GeorgianHouse();  
    House house = ghouse;  
    System.out.println(ghouse.className);  
    System.out.println(house.className);  
    ghouse.showName();  
    house.showName();  
}
```

GeorgianHouse

House



The diagram consists of two curved blue arrows. One arrow originates from the text 'GeorgianHouse' and points to the text 'House'. The second arrow originates from the 'house.className' property access in the main method of the code and points to the 'House' text, illustrating that the runtime class of the 'house' object is 'House'.

Something More Strange

58

```
public class Father {  
    String name = "Father";  
  
    public void showName() {  
        System.out.println(name);  
    }  
}
```

```
public class Son extends Father {  
  
    public String name = "Son";  
  
    public static void main(String[] args) {  
        Father person = new Son();  
        person.showName();  
    }  
}
```

What is the result?

```
public class Father {  
    String name = "Father";  
  
    public void showName() {  
        System.out.println(name);  
    }  
}
```

```
public class Son extends Father {  
  
    public String name = "Son";  
  
    public void showName() {  
        System.out.println(name);  
    }  
  
    public static void main(String[] args) {  
        Father person = new Son();  
        person.showName();  
    }  
}
```



What is the result?

```
public class Father {  
    String name = "Father";  
  
    public void print() {  
        System.out.print(name);  
    }  
  
    public void showName() {  
        print();  
    }  
}  
  
public class Son extends Father {  
  
    public String name = "Son";  
  
    public void print() {  
        System.out.print(name);  
    }  
  
    public static void main(String[] args) {  
        Father person = new Son();  
        person.showName();  
    }  
}
```

What is the result?

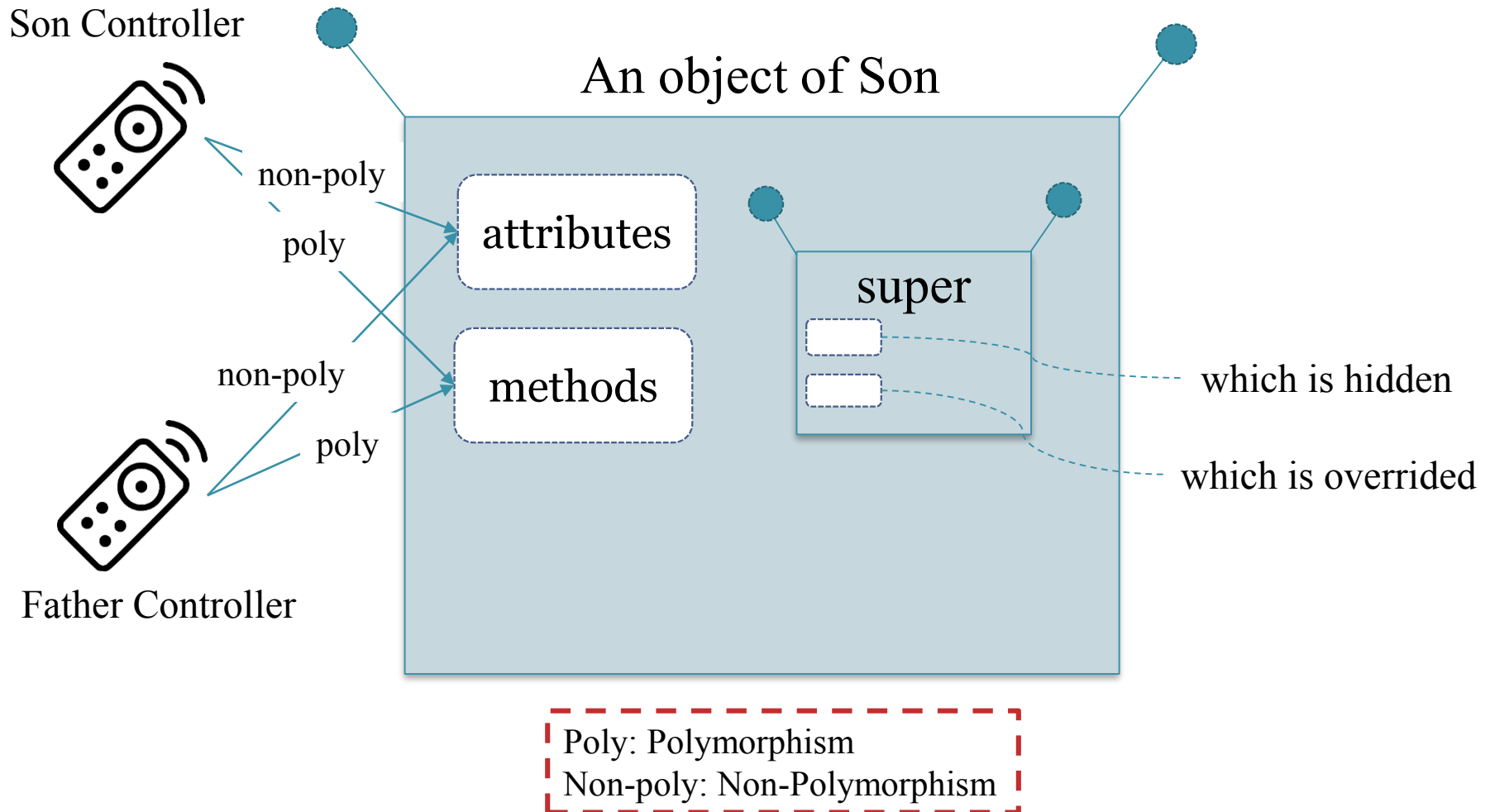
```
public class Father {  
    String name = "Father";  
    private void print() {  
        System.out.print(name);  
    }  
  
    public void showName() {  
        print();  
    }  
}
```

```
public class Son extends Father {  
  
    public String name = "Son";  
    public void print() {  
        System.out.print(name);  
    }  
  
    public static void main(String[] args) {  
        Father person = new Son();  
        person.showName();  
    }  
}
```

What is the result?

A Model of Inheritance and Polymorphism

62



Forecast

63

- A Notion of Exception
- Java Exceptions
- Exception Handling
- User-defined Exceptions
- How to Use Exception