Chapter 2: Java OO II

XIANG ZHANG

javaseu@163.com

https://wdsseu.github.io/java/

Content

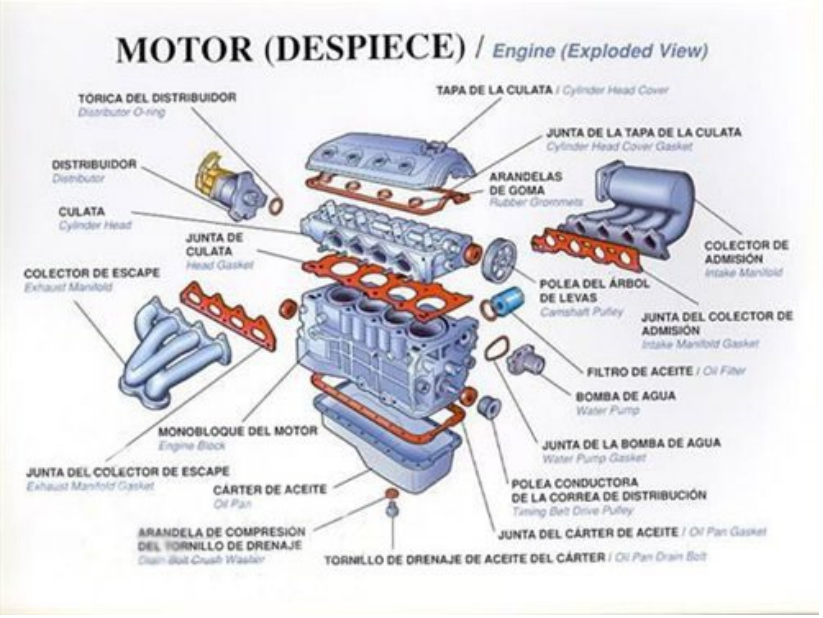
- Abstraction
  - Abstract Class
  - Interface
- Inheritance
- Polymorphism

Abstraction

Abstraction

- 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



Abstraction – Abstract Class

- 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

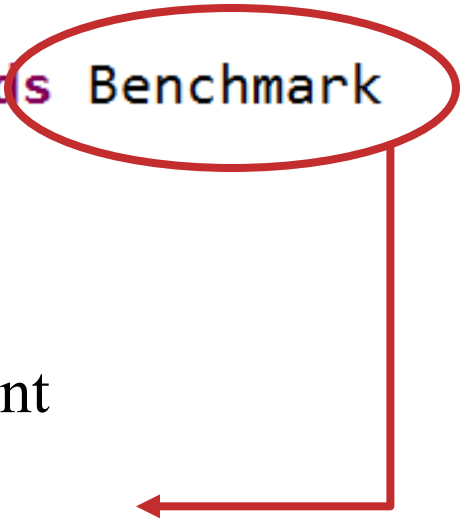
Abstraction – Implemented Class

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

- 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 two sorting algorithms;
  - judging the correctness of sorting;
  - comparing your time efficiency to the efficiency of java sorting algorithm

Lab Work

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

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

Abstraction – Interface

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

Abstraction – Interface

- 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


```
public interface Tweetable {

    public void tweet();

}
```

```java
public class Sparrow implements Tweetable {

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

}
```

Tweetable

```java
public class Hummingbird {

    public void fly(Place a, Place b){

    }

}
```

Non-tweetable

Abstraction – Interface

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

Why variables are not allowed in an interface?

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

Abstraction - Interface

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

```java
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

- Fields in Interface
  - public
  - static
  - final
  - Must be initialized

why static?

```
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!
        }
}
```

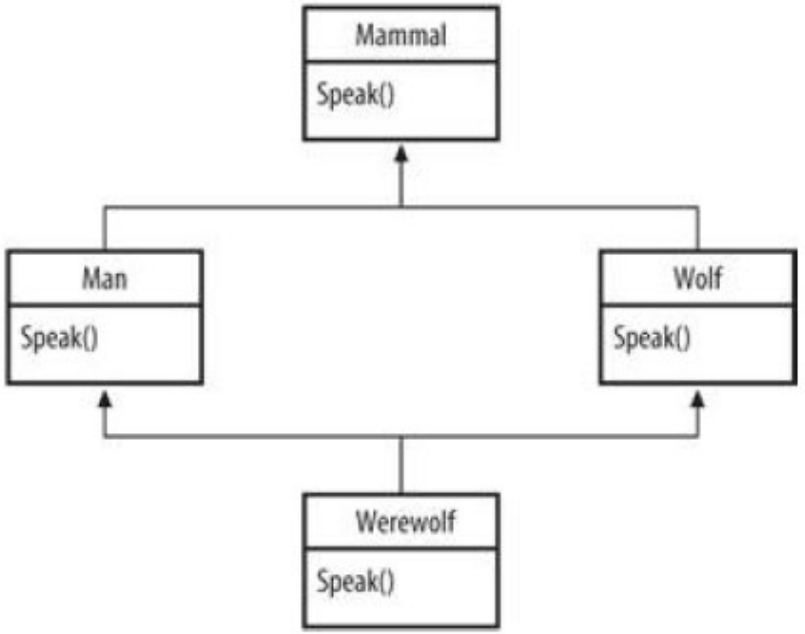
Abstraction – Interface

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

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

Abstraction – Interface

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



Abstraction – Interface

- 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

- Interface-based Programming
  - Agile: First, lets 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

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

Abstraction – Interface

- 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

- Combining Abstract Class and Interface

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

Inheritance

Inheritance

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

Inheritance – Class

- Example - House

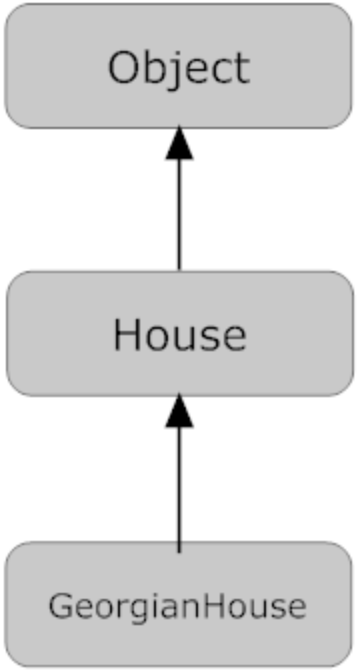
```
public class House {
    private String doorStyle;
    private String windowsStyle;
    private String wallStyle;

    public House(String door, String windows, String wall){
        doorStyle = door;
        windowsStyle = windows;
        wallStyle = wall;
    }

    // Getters and Setters goes here...
}
```

Inheritance – Class

- 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



Inheritance – Class

- 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

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

Inheritance – Class

- 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

- 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?


```
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

- Limits of Overriding
  - In overriding, the access rights of a method in subclass could be unchanged， enlarged or reduced???
  - In overriding, the return type of a method in subclass could be unchanged ， enlarged or reduced ???

```
Class Father{
    protected Father test(){…}
}

Class Son extends Father{
    public Father test(){…}
}  //enlarged
```

```
Class Son extends Father{
    protected Father test(){…}
}  //unchanged

Class Son extends Father{
    private Father test(){…}
}  //reduced
```

```
Class Father{
    protected Father test(){…}
}

Class Son extends Father{
    protected Object test(){…}
}  //enlarged
```

```
Class Son extends Father{
    protected Father test(){…}
}  //unchanged

Class Son extends Father{
    protected Son test(){…}
}  //reduced
```

Inheritance – Class

- 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 sublcass should $\subseteq$ A
    - If return type in superclass is a primary type, it should be unchanged in subclass.

Inheritance -Class

- 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

- 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

- 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

- 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


```
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();
}
```

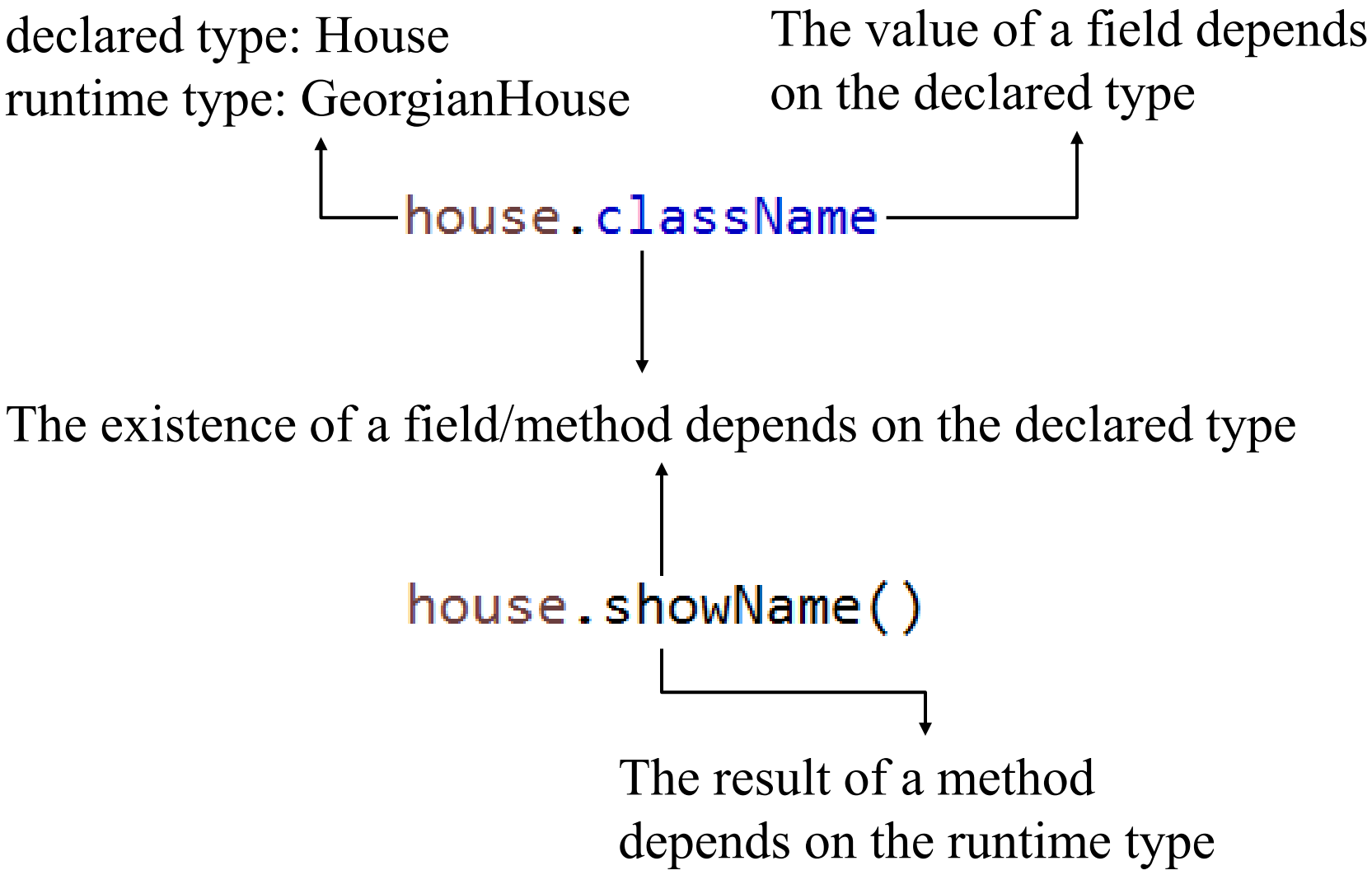
A:
GeorgianHouse
House
The extended class: GeoriganHouse
The extended class: GeoriganHouse

B:
GeorgianHouse
GeorgianHouse
The extended class: GeoriganHouse
The extended class: GeoriganHouse

C:
GeorgianHouse
House
The extended class: GeoriganHouse
The super class: House

D:
GeorgianHouse
House
The extended class: GeoriganHouse
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

- 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

- 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

- 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

- 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

- 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

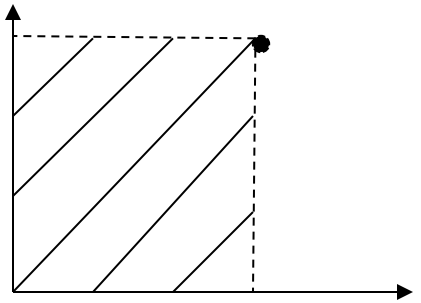
- Multiple Inheritance for Interfaces

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

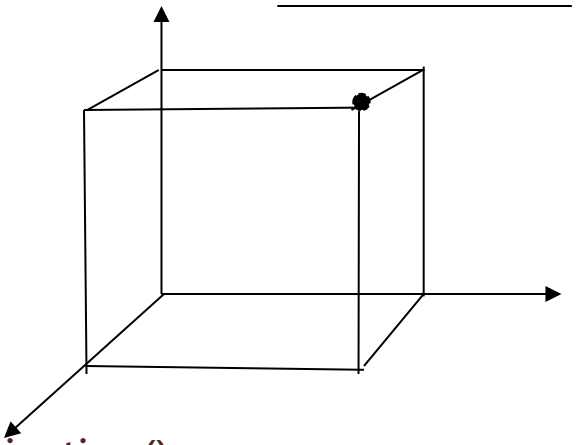
- Inheritance of Interfaces also has:
  - Hiding of fields
  - Override of Methods

Lab Work

- Define a superclass: 2DPointer
  - x, y // x value, y value;
  - constructions
  - distance() //distance to ground zero;
  - projection() // the size of the shadow area



- Define a subclass 3DPointer
  - x, y // x value, y value;
  - constructions
  - distance() // distance to ground zero;
  - projection() // the volume of the shadow area



- A Tester Class
  - create a 3Dpointer and test its distance() and projection()

Polymorphism

Polymorphism

- 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

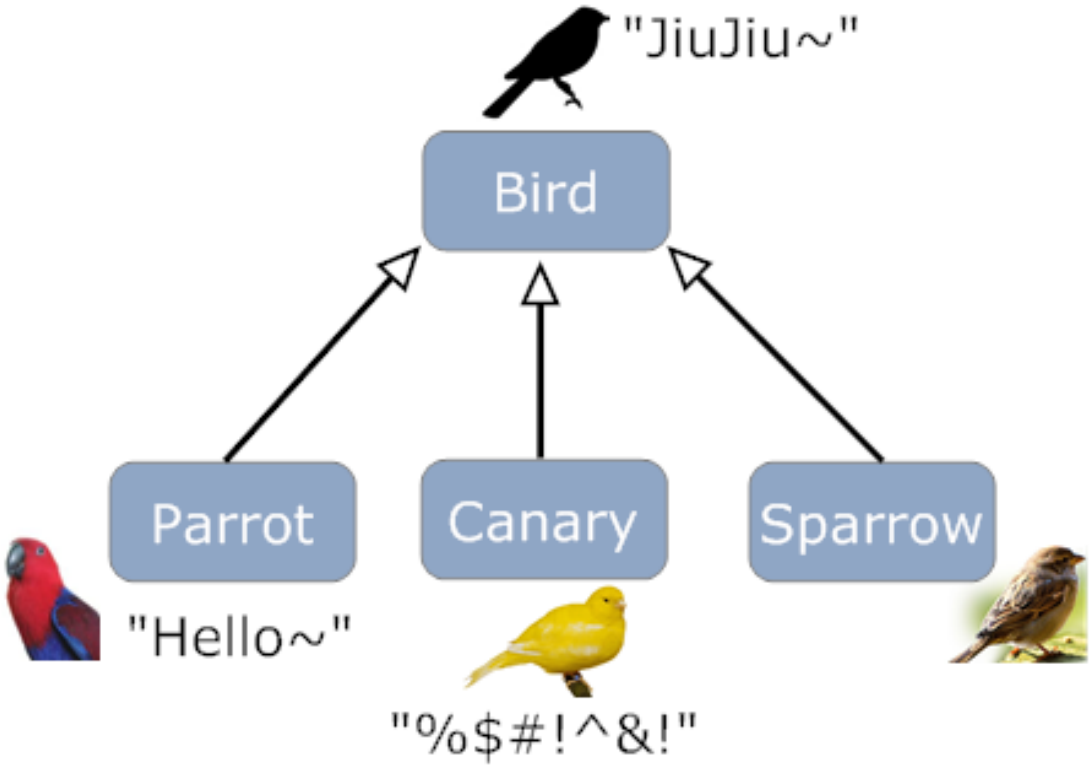
- Static Polymorphism
  - Polymorphism which can be determined in compilation
  - Overloading

```
Calculator.add(10, 9)
Calculator.add(0.5, 0.4)
```

Polymorphism

- 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



Polymorphism

- 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

- Interface Polymorphism

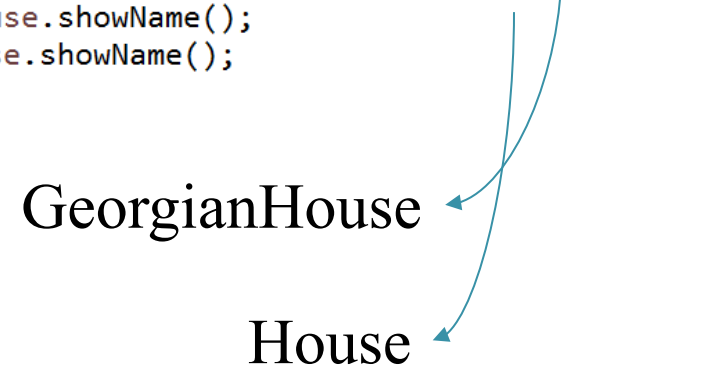
```
public interface Tweetable{
    public void tweet();
}

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


```
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();
}
```



Something More Strange


```
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?

Principle of Proximity (就近原则)


```
public class Actor {
    private String name;

    public Actor(String name){
        this.name = name;
    }

    public void showName(){
        String name = "周润发";
        System.out.println(name);     //周润发
        System.out.println(this.name); //范冰冰
    }

    public static void main(String[] args){
        Actor a = new Actor("范冰冰");
        a.showName();
    }
}
```

```java
public class Father {
    String name = "Father";

    public void showName() {
        System.out.println(name);
    }
}
```

```java
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?

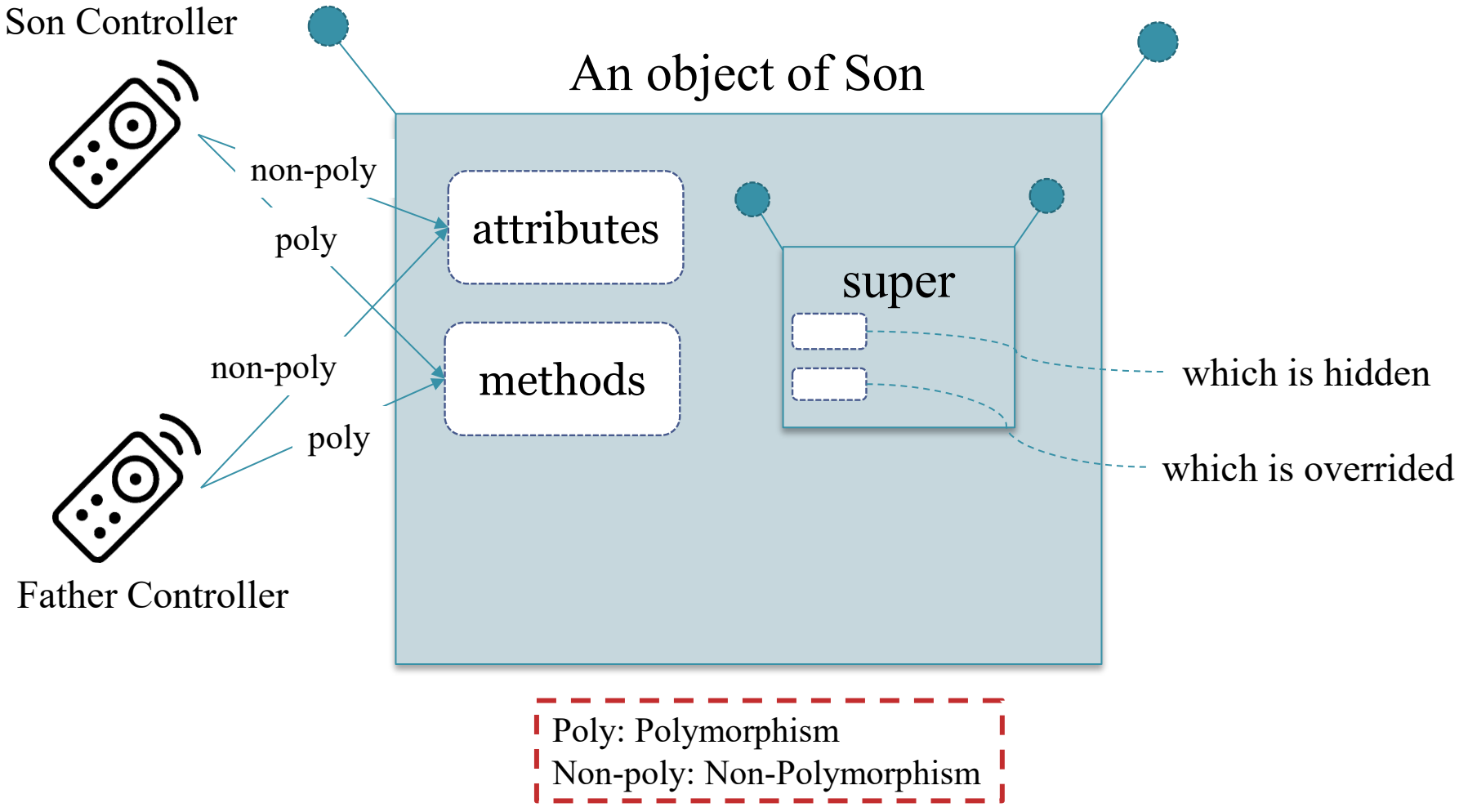
```java
public class Father {
    String name = "Father";

    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?

```java
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



Forecast

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