

# 面向对象期末复习

---

## 面向对象期末复习

### 一、类与对象

1. 类
2. 对象
3. 实例变量，又称成员变量
4. 静态变量
5. 实例方法，又称成员方法
6. 静态方法，又称类方法：
7. 构造器

### 二、继承

1. 定义
2. 范围修饰符
3. `this` 关键字
4. `super` 关键字

### 三、多态

1. 定义
2. 动静绑定
3. 实例方法：动态绑定
4. 静态方法：静态绑定
5. 实例、静态变量
6. `instanceof`

### 四、`final` 关键字

1. `final` 变量
2. `final` 方法
3. `final` 类

### 五、抽象类

### 六、接口

### 七、其他

1. `package` 语句
2. 枚举
3. 泛型
  - 3.1: 泛型参数
  - 3.2: 泛型方法
  - 3.3: 泛型类
  - 3.4. 向上转型

# 一、类与对象

## 1. 类

- 一个模板，描述一类对象的行为和状态

## 2. 对象

- 类的实例，有自己的参数，可以调用类中的方法

## 3. 实例变量，又称成员变量

- 每个对象的属性
- 每个对象的实例变量的值可以不同
- 调用格式
  - 此对象：`this.<变量名>`，无变量名冲突时可直接使用 `<变量名>`
  - 其他对象：`<对象名>.<变量名>`
- has-a relation：一个类把其他类的对象作为实例变量，自然语言语法为 `<类名> has a/an <实例变量类名>`

复合

```
public class House {  
    kitchen kitchen;  
    int size;  
}
```

上述代码存在 has-a 关系：House has a Kitchen

## 4. 静态变量

- 静态变量由这个类的所有成员共享
- 调用格式
  - 本类内部：`<类名>.<静态变量名>` 或 `<任意对象名>.<静态变量名>`，无变量名冲突时可直接使用 `<静态变量名>`
  - 本类外部：仅 `<类名>.<静态变量名>` 或 `<任意对象名>.<静态变量名>`

不能用 this 调

## 5. 实例方法，又称成员方法

- 调用格式
  - 此对象：`this.<实例方法名>`，或直接使用 `<实例方法名>`
  - 其他对象：`<对象名>.<方法名>`
- 必须在创建对象后才能使用
- 每个对象的实例方法运行结果可能不同

## 6. 静态方法，又称类方法：

- 调用格式
  - 本类内部：<类名>.<静态方法名> 或 <任意对象名>.<静态方法名>，或直接使用 <静态方法名>
  - 本类外部：<类名>.<静态方法名> 或 <任意对象名>.<静态方法名>
- 可以在没有创建对象时调用
- 规定：主方法为静态方法
- 静态方法内部不可调用实例变量和实例方法，因此不能出现任何 `this` `super` 关键字

可用对象名调

	可调用静态变量、方法	可调用实例变量、方法
静态方法	√	
实例方法	√	√

## 7. 构造器

- 可以重载：一个类中可以有多个方法签名不同的构造器
- 默认构造器
  - 触发条件
    - 没有在类中显式创建构造器时，编译器会自动加一个默认构造器
    - 只要类中有自己写的构造器，无论是有参还是无参，都不会再生成默认构造器。

(一个 constructor 都不写)

### 参数

- 无传入参数
- 将基本类型初始化为默认值
- 将引用类型初始化为 `null`

可能出错

- 枚举里的构造器必须 `private` 或者 `no-modifier`
- 构造器返回的是一个对象的引用，即 内存地址

构造器传递内存地址

问：下述代码的运行流程是什么？

(即赋值时)

赋新 new 对象的地址

```
import java.util.ArrayList;

public class test {
    public static void main(String... args) {
        ArrayList<Integer> list = new ArrayList<>();
        // 构造器返回内存地址，该值被赋值给变量 list

        list.add(0);

        addElement(list);
        // list 的值通过方法调用传入 addElement 方法
        // 方法调用后，链表内容为 {0, 1}

        System.out.println(list.get(1));
        // 输出: 1
    }
}
```

传地址复制  
list 指向 list  
new list

```
public static void addElement(ArrayList<Integer> list){
    // addElement 方法接收到值后，新建变量 list，将值赋值给此变量

    list.add(1);
    // 此时 list 依然是主方法中的内存地址，add 操作针对的是主方法中创建的链表
    // {0, 1} 对同一个地址操作

    list = new ArrayList<>();
    // 构造器返回了一个新的内存地址，覆盖掉了原本来自主方法的内存地址
    // 后续的 add 操作针对的是新内存地址
    list.add(2);
    list.add(3);
    // 方法体结束，新内存地址被回收
}
}
```

## 二、继承

### 1. 定义

- 语法：使用 `extends` 关键字连接两个类，如 `class B extends A {...}`
- `B extends A`  $\iff$  `B is a/an A`
  - 别名：is-a 关系，
- 子类可以重写父类中的方法
  - 重写：在子类中重新写一遍与父类中相同方法签名的方法，使父类对象和子类对象具有不同功能

### 2. 范围修饰符

- 子类继承父类的修饰符后可以改大，不能改小

	本类中	同一个包下	这个类的子类	所有地方
public	√	√	√	√
protected	√	√	√	
no modifier	√	√	同包中子类可访问 不同包子类不可访问	
private	√			

### 3. this 关键字

- 表示对象：
  - `this` 只能表示本类的对象，方法被谁调用，`this` 就表示谁
  - 可以用 `this.<变量名>` 和 `this.<方法名>` 来调用实例变量和实例方法
  - 传入参数与本类实例变量有相同名字时，需要用 `this.<变量名>` 表示实例变量
- 表示构造器：
  - 构造器中可使用 `this(...)` 来调用本类的另一个构造器

- 该 `this` 语句必须是这个构造器的第一条语句
- 问：
  - 调用静态变量有一种语法是 `<对象名>.<静态变量名>`，那么是否可以使用 `this.<静态变量名>` 来调用？
  - 答：不一定。

静态方法中，由于静态变量需要被允许在静态方法中调用，静态方法内不允许使用 `this` 关键字

**静态方法中不存在“当前实例”**

实例方法中，可以使用 `this` 关键字，因此 `this.<静态变量名>` 符合 `<对象>.<静态变量名>` 的语法

## 4. `super` 关键字

- 调用父类中的方法： `super.<方法名>`
- 调用父类构造器：
  - 创建子类对象时，必须先创建一个父类对象，即子类构造器中必须先调用父类构造器
  - 如果不显式调用：
    - 默认调用父类的无参构造器，注意默认构造器也是无参构造器
  - 如果父类没有无参构造器则报错
  - 如果显式调用：
    - 用 `super(...)` 调用父类构造器，括号内为父类构造器传入参数
    - 该语句必须是子类构造器的第一条语句
- constructor chain:
  - 现有继承关系： `Object ← A ← B ← D ← E`，假如需要创建 E 类对象，则构造器内如何调用？
  - 想要新建子类对象，必须先新建一个父类对象
  - 所有类都是 `Object` 的子类
  - 因此，无论要创建哪个类的对象，最开始被创建的一定是 `Object` 对象

## 三、多态

### 1. 定义

- 父类引用指向子类对象
- 隐含条件：必须有父子类关系
  - “父子类关系”可以是 `extends` 具体类或抽象类，也可以是 `implements` 接口
- 例： `Father father = new Child();`
  - `Father` 被称为“引用类型”
  - `father` 被称为“引用”
  - `new Child()` 被称为“对象”
- 一般子类会重写父类中的方法

## 2. 动静绑定

仅多态+实例方法

- 动态绑定：编译阶段不知道调用哪个方法，运行时才知道
- 静态绑定：在编译阶段就知道调用什么方法/变量

## 3. 实例方法：动态绑定

引用类

child.<实例方法名>，根据以下流程判定调用哪个方法：

- 在父类中检查是否有该方法签名
- 如果父类中没有此方法签名：
  - 编译错误
- 如果父类中有此方法签名：
  - 调用子类中的该方法，去子类中检查是否有该方法签名
  - 如果子类重写方法，则调用重写后的方法
  - 如果子类没有重写，则调用从父类继承来的方法

编译时仍认为是父类对象

## 4. 静态方法：静态绑定

override

- final static private 方法都是静态绑定，始终调用等号左边的父类中的方法。

特注：权限修饰符

## 5. 实例、静态变量

- 由等号左边的父类决定
- 注：继承关系中，子类可以继承父类中的变量，但多态中只看父类

静态

## 6. instanceof

关键词而非方法

- 用于检查一个引用是否是某个类的对象
- 语法：if (<对象名> instanceof <类名>) {...}
- 如果 <对象> 是 <类名> 的实例则为 true，否则为 false
- 子类对象 instanceof 父类 返回值为 true

对象 object	类名 className	返回值
父类	父类	true
父类	子类	false
子类	父类	true
子类	子类	true

引用类型

- 强调：多态中，对象类型位于等号右侧，而非等号左侧。

◦ 即：对于 Father obj = new Child() 而言

- obj instanceof Child 的返回值为 true
- obj instanceof Father 的返回值为 true

instanceof 判断实例到底什么类型

- 应用：向下转型。先用 instanceof 检查一个引用是否指向正确的对象，再向下转型

## 四、final 关键字

### 1. final 变量

- 必须被赋值
- 被赋值后无法更改
- 要么<sup>①</sup>声明变量时赋值，要么在所有构造器里赋值<sup>②</sup>

### 2. final 方法

- 父类中的 final 方法无法在子类中被重写

### 3. final 类

- final 类无法被继承
- 一旦写好这个类后就无法改变，例子有 String 类

## 五、抽象类

1. 抽象类不能用于实例化对象

2. 一般使用方法：

- 子类继承抽象类，重写抽象方法
- 如果子类没有重写完所有抽象方法，则子类也必须声明为抽象类
- 用抽象类的非抽象子类实例化对象

但抽象类可有构造器

3. 抽象方法不能为 private — 不可 override

4. 有抽象方法一定是抽象类，但抽象类不一定有抽象方法（抽象类可以拥有具体方法）

5. 可以有构造器和静态方法，但均不能声明为 abstract

用假设法思考

- 原因：构造器不能被继承，静态方法可以被继承但是不能被重写

6. 抽象类可以作为多态中的父类

若“最终”仍有

7. 可通过 <抽象类名>.<静态方法名> 调用抽象类中的静态方法

抽象类 override → CE

8. 抽象类不能用 final 声明

- 原因：final 声明的类无法被重写

## 六、接口

1. 语法：<类名> implements <接口名>

```
public interface Payable {
    double getPaymentCount();
}

public class Employee implements Payable {
    double getPayableCount() {
        // .....
    }
}
```

2. 变量默认且必须是 `public static final`，必须被初始化

3. 方法默认且必须是 `public abstract`

4. 接口不能拥有构造器，不能被实例化

完全抽象

5. 对比：

- 一个类可以 `implements` 任意多个接口，只能 `extends` 一个父类
- 一个接口可以 `extends` 任意多个接口，不能 `implements` 接口或类
- 接口的继承关系与类的继承关系是两个不同的体系，此处死记硬背即可

6. 接口可以作为多态中的父类

7. Java 8 引入了一种新特性：接口中可以存在由 `static` 或 `default` 修饰的具象方法，但是没学，考试不考

## 七、其他

### 1. package 语句

- 放在 `.java` 文件首行，用于声明文件位置

### 2. 枚举

- 一种很高级的常量
- 每个变量都为 `final` 和 `static`
- 枚举对象的构造器必须为 `private`
- 枚举是一种类，可以重载构造器

### 3. 泛型

#### 3.1：泛型参数

- 需要满足变量命名规则（字母数字下划线美刀）。理论上可以用任何合法字符串作为变量，但有几个约定俗成的名字：
  - `T`：引用数据类型，不能是八大基本类型
  - `E`：集合中存放的元素
- 泛型参数是数据类型的代称，运行时会变成传入的数据类型。
- 只能表示引用类型，无法表示基本数据类型。

(包装)

#### 3.2：泛型方法

- 命名规范及举例

方法声明	是否合法	说明
<code>public static &lt;T&gt; void printArray(T[] array)</code>	合法	所有关键字都出现
<code>&lt;T&gt; void printArray(T[] array)</code>	合法	默认范围修饰符，实例方法



方法声明	是否合法	说明
<u>&lt;T&gt;</u> static printArray(T[] array)	不合法	<T> 必须在 static 和 范围修饰符 之后
<T> void printArray(int[] array)	合法	可以声明泛型参数但不使用
<K, V> void printPair(K[] arrayK, V arrayV)	合法	多个泛型参数用 <u>逗号分隔</u>
<hdfjka> void printArray(hdfjka[] array)	合法	泛型参数命名符合标识符规则

- 下面的代码中，传进来的数组是什么引用类型，执行的时候 T 就变成什么类型

```
public static <T> void printArray(T[] array) {
    for (T element : array)
        System.out.printf("%s ", element);
    System.out.println();
}
```

- Bounded Type Parameter (有界类型形参)
  - 尖括号中为标记符的限制条件，只有符合条件的类型才能替代标识符
  - 尖括号里的 extends 实际上表示 extends 或 implements
  - 举例：下面的代码中，只有使用了 Comparable 接口的引用类型才能作为 T

```
public static <T extends Comparable<T>> void printArray(T[] array) {
    for (T element : array)
        System.out.printf("%s ", element);
    System.out.println();
}
```

### 3.3: 泛型类

- 在类名后面添加了类型参数声明
- 写类中的代码时可以用 T

在类名后

```
public class Box<T> {

    private T t;

    public void add(T t) {
        this.t = t;
    }

    public T get() {
        return t;
    }

}
```

### 3.4. 向上转型

- 假设 `G` 为泛型类，现有继承关系 `Child extends Father`，则 `G<Child>` 不是 `G<Father>` 的子类。

```
String s = "1234";
Object o = s;
// String 是 Object 的子类，可以正常运行

ArrayList<String> list1 = new ArrayList<>();
List<String> list2 = list1;
// ArrayList<String> 是 List<String> 的子类，可以正常运行

ArrayList<String> strList = new ArrayList<>();
ArrayList<Object> objList = strList;
// 报错，ArrayList<String> 不是 ArrayList<Object> 的子类
```