方法重载 Method Overloading

- **方法重载 Overloading 是指：在同一个类中**，多个方法的**方法名相同**，但**参数不同（数量不同 /类型不同）**

▼ 方法重载的条件　　❗必须满足

1. **方法名**要相同
2. 参数**个数**不同 / 参数**类型**不同
3. 返回值可以相同也可以不同, 但不能**只靠返回值不同**来区分❗

也就是说, **方法名要一样, 参数要有点不一样, 返回值无所谓**

▼ 代码实现

```
public class Calculator {

    // 加法：两个整数
    public int add(int a, int b) {
        return a + b;
    }

    // 加法：三个整数
    public int add(int a, int b, int c) {
        return a + b + c;
    }

    // 加法：两个小数
    public double add(double a, double b) {
        return a + b;
    }
```

```
    public static void main(String[] args) {
        Calculator calc = new Calculator();
        System.out.println(calc.add(2, 3));        // 调用第一个
        System.out.println(calc.add(2, 3, 4));     // 调用第二个
        System.out.println(calc.add(2.5, 3.1));    // 调用第三个
    }
}
```

输出结果：

```
5
9
5.6
```

▼ 常见误区 👉 ❌ **返回值不同不能构成重载**

```
public int getValue(int a) { return a; }
public double getValue(int a) { return (double)a; } // ❌ 报错：重复定义
```

▼ **方法重载的参数匹配规则**

```
cat.f(1);

void f(int i)      { }  // 基本类型 int
void f(Integer i)  { }  // 包装类型 Integer
void f(Number i)   { }  // 父类类型 Number（Integer 的父类）
```

✅`f(1)` **最终会调用调用** `f(int i)`

▼ 📌 Java 编译器如何选择重载方法？

当写 `cat.f(1)`，这个 `1` 是**字面量整数**，类型是 `int`，Java 编译器会按以下顺序寻找最合适的方法：

1. **精确匹配**：找参数正好是 `int` 的方法 ➜ `f(int)` ✅
2. **自动装箱（boxing）**：找参数是 `Integer` 的方法 ➜ `f(Integer)`
3. **类型提升（widening）**：找参数是 `long`、`float` 等 ➜ 不适用
4. **父类 / 接口类型**：如 `f(Number)`
5. **可变参数 varargs**（可变参数最后一选） ➜ 这里没有用

- 所以 `1` 是 `int`
- 先找 `f(int)`，找到了 ✅，直接调用它
- 不会去找 `Integer` / `Number`

🤔 如果 `f(int)` 被注释掉呢？

```
// void f(int i) { }
```

再调用 `cat.f(1)`

👉 Java 会做 **自动装箱**，把 `int 1` 转为 `Integer`，于是调用：

```
void f(Integer i) { }
```

😲 如果把 `f(int)` 和 `f(Integer)` 注释掉，只剩下 `f(Number)` 呢？

那么 `int 1` 会：

- 自动装箱成 `Integer`
- 再自动提升为 `Number`
- 最终调用 `f(Number)` ✅

> Java 选择重载方法时的顺序：

> **精确匹配变量声明的类型 → 自动装箱/拆箱 → 类型提升（父类）→ 可变参数**

▼ null 没有类型, 如何匹配❓

- 这是 Java 重载判断里**最容易出错**、也最考验理解的问题之一
- 🔍 先明确一点👉 null ➜ 可以赋值给任何引用类型（对象）, 但它本身没有类型
- ✅ Java 编译器会根据 **most specific type 来决定**调用哪个方法

```
cat.f(null);

void f(Integer i) { ... }
void f(Number i)  { ... }
```

- `Integer` 是 `Number` 的**子类，更具体**
- 所以 Java 会调用 `f ( Integer i )`

▼ 如果写了两个**没有继承关系**的重载方法 👉 报错❌

```
void f(String s)  { ... }
void f(Integer i)  { ... }
```

然后调用：

```
cat.f(null);
```

这时候就 **编译错误**❌ 了！

> 编译器无法判断 null 应该调用 f(String) 还是 f(Integer)，因为它们同样具体，没有父子关系

```
❌ 错误信息会类似：
reference to f is ambiguous
```

```
both method f(java.lang.String) and method f(java.lang.Integer) match
```

✅ 总结

方法签名	`cat.f(null)` 调用哪个？
`f(Object)` 和 `f(String)`	调用 `f(String)`（更具体）
`f(Number)` 和 `f(Integer)`	调用 `f(Integer)`（更具体）
`f(String)` 和 `f(Integer)`	❌ 编译错误，调用不明确（ambiguous）

> null选子类，选最具体；两个不相关，就报错

▼ 构造器的重载 Constructor Overloading

- 构造器的重载和方法的重载是**一样的规则** 👉 不过它用在**创建对象的时候**
- Constructor 是创建对象时被调用的“初始化函数”，没有返回值，方法名与类名相同

▼ 构造器重载

```
public class Person {                类
    String name;                  成员变量
    int age;                    成员变量

    // 构造器1：无参
    public Person() {
        this.name = "Unknown";
        this.age = 0;
    }

    // 构造器2：只传名字
    public Person(String name) {
        this.name = name;
        this.age = 0;
    }
```

```
    // 构造器3：传名字和年龄
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public void introduce() {               成员方法
        System.out.println("Hi, I'm " + name + " and I'm " + age + " years
old.");
    }


    public static void main(String[] args) {
        Person p1 = new Person();                    // 用无参构造器
        Person p2 = new Person("Alice");              // 用一个参数构造器
        Person p3 = new Person("Bob", 25);             // 用两个参数构造器

        p1.introduce();  // Hi, I'm Unknown and I'm 0 years old.
        p2.introduce();  // Hi, I'm Alice and I'm 0 years old.
        p3.introduce();  // Hi, I'm Bob and I'm 25 years old.
    }
}
```

▼ 构造器重载 + 构造器互相调用

```
public class Cat {
    int age;
    String name;

    // 无参构造：默认1岁，名字叫“张三”
    Cat() {
        this("张三");  // 调用下面那个只有name参数的构造方法
    }
```

```
    // 只有名字参数：默认1岁
    Cat(String name) {
        this(1, name); // 调用下面这个 age+name 的构造方法
    }

    // 年龄和名字都可以指定
    Cat(int age, String name) {
        this.age = age;
        this.name = name;
    }

    public static void main(String[] args) {
        new Cat();  // 触发构造链条
    }
}
```

在 `main` 中写的是：

```
new Cat();
```

Java 一步步执行的逻辑：

1. 调用 `Cat()`（无参构造器）
2. → 它调用 `this("张三")`
3. → `Cat(String name)` 被调用
4. → 它又调用 `this(1, name)`
5. → 最终执行 `Cat(int age, String name)`，将 `age=1`、`name="张三"` 赋值给成员变量

所以最后这只猫的属性

```
age = 1;
name = "张三";
```

🟡 `this(...)` 和 `this.xxx = xxx` 有什么区别？

用法	作用
`this(...)`	调用**另一个构造方法**
`this.变量名 = 值`	给**当前对象的属性赋值**

🔥 注意事项：构造器中，`this(...)` 一定要放在第一行

```
Cat(String name) {
    this(1, name);      // ✅ 正确
    // this.age = 1;    // ❌ 如果放在 this(...) 之前，会报错
}
```

▼ **构造方法重载 + this(...) 链式调用 的意义**

- ✅ **让代码更清晰、更复用、更灵活，更易维护**

▼ 减少重复代码（代码复用）

❌ 如果不用 `this(...)`，每个构造方法都要重复写初始化逻辑：

```
Cat() {
    this.age = 1;
    this.name = "张三";
}

Cat(String name) {
    this.age = 1;
    this.name = name;
}

Cat(int age, String name) {
    this.age = age;
    this.name = name;
}
```

每个方法里都要写 `this.age = 1`、`this.name = xxx`，容易出错、难维护

✅ 用 `this(...)` 链式调用，逻辑只写一遍：

```
Cat() {
    this("张三");
}

Cat(String name) {
    this(1, name);
}

Cat(int age, String name) {
    this.age = age;
    this.name = name;
}
```

只在最底层的构造器中写一次赋值逻辑，**其他构造器都可以用它来完成工作**

以后要改逻辑（比如默认年龄从 1 改成 3），只改一处就够了

▼ 提供更灵活的使用方式（给用户多个选择）

现在的构造器重载写法支持：

用法	效果
`new Cat()`	默认：1岁，张三
`new Cat("小黑")`	1岁，小黑
`new Cat(3, "小花")`	3岁，小花

程序使用者可以选择需要传几个参数，不同需求都能满足 👉 **灵活性**

▼ 以后会写很多“工具类” / “实体类”，都需要构造方法重载

- `new User()`：创建一个空用户
- `new User("lyra")`：只设用户名
- `new User("lyra", "123456")`：设置用户名+密码

这样用起来才方便 👉否则就只能死板地每次都传一堆参数

▼ **重载 Overload 🆚 重写 Override**

对比项	重载 Overloading	重写 Overriding
出现位置	同一个类中	父类和子类之间
方法名	必须相同	必须相同
参数列表	必须不同（个数或类型不同）	必须**完全相同**
返回类型	可以相同也可以不同	**必须相同**（或是返回类型的子类）
访问修饰符	没有限制	子类中的方法访问权限**不能更低**
关键词	无需使用任何关键词	子类中建议使用 `@Override` 注解
编译时 or 运行时	编译时决定调用哪个方法	运行时决定调用哪个方法（多态）
用途	提高方法的灵活性，功能扩展	子类根据自己的需要**修改父类的方法**

> 重载看参数，重写看类层
>
> ➤ 参数不同是重载，
>
> ➤ 子类覆盖父类是重写

▼ ✅ 方法重写　　子类改写父类

```
class Animal {
    public void speak() {
        System.out.println("动物在叫");
    }
}

class Dog extends Animal {
    @Override
    public void speak() {
        System.out.println("狗在汪汪叫");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal a = new Animal();
```

```
        Dog d = new Dog();
        Animal ad = new Dog();  // 向上转型，多态

        a.speak();   // 输出：动物在叫
        d.speak();   // 输出：狗在汪汪叫
        ad.speak();  // 输出：狗在汪汪叫（重写后调用子类方法）
    }
}
```

▼ ✅ 方法重载

```
public class Printer {
    public void print(String text) {
        System.out.println("打印字符串：" + text);
    }

    public void print(int number) {
        System.out.println("打印整数：" + number);
    }

    public void print(String text, int times) {
        for (int i = 0; i < times; i++) {
            System.out.println("打印：" + text);
        }
    }
}
```

如果你想练习区分这些概念，我可以出几道题目让你判断是“重载”还是“重写”，要来一组小测试吗？✅