

第22课

同一命令 不同攻击

用父类合同指挥不同对象

引用找到对象，对象提供重写行为

Enemy enemy



Skeleton 对象

attack() 返回射箭描述

指挥官只喊一次 attack，僵尸挥击、骷髅射箭、蜘蛛扑咬。跟着引用找到真实对象，你会看懂 Java 如何选中正确方法，并完成混合波次。

前置背包：第21课的 Enemy 家族、extends、super 和方法重写。今天让整支怪物波次听同一条命令。

阅读方式：先猜结果，再运行核对；每次只改一个条件。

本章使用 JDK 21，无需预览功能。故事与插画借用方块世界，练习运行在普通 Java 控制台。

开始前：先预测，再让程序回答

本章使用 JDK 21，无需预览功能。故事借用方块世界，示例运行在普通 Java 控制台；数值和规则若是课程模型，都会明确说明。

每个普通示例都是完整独立文件，按框上标注的文件名保存。各示例放在不同文件夹，避免辅助类重名。完整项目分段时，按顺序接在同一个文件中。

终端命令：中文占位符换成实际文件名与类名

```
javac -encoding UTF-8 示例文件名.java  
java 示例类名
```

- 预测：先写预期输出，或画清引用和数据的变化。
- 验证：运行后逐行核对，再只改一个条件。
- 解释：说清结果来自 Java 规则、课程合同还是输入数据。

“故意错误”用于认识 Bug，不能与普通示例混用。文件实验使用专门临时目录；不要拿真实存档测试覆盖和删除。

01 父类引用容纳子类对象

Enemy enemy = new Skeleton(...) 左边说明程序通过 Enemy 合同使用它，右边决定真正创建的是 Skeleton。

引用找到对象，对象提供重写行为

Enemy enemy



Skeleton 对象

attack() 返回射箭描述

引用的声明类型是 Enemy，运行时对象类型是 Skeleton 或 Zombie。两者同时存在，不要混成一个概念。

L22Example01.java • 完整独立示例

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " 普通攻击"; }
}
class Skeleton extends Enemy {
    public String reload() { return name + " 补充箭矢"; }
    Skeleton(String name) { super(name); }
    @Override public String attack() { return name + " 射箭"; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " 挥击"; }
}
public class L22Example01 {
    public static void main(String[] args) {
        Enemy a = new Zombie("腐肉");
        Enemy b = new Skeleton("小白");
        System.out.println(a.attack());
        System.out.println(b.attack());
    }
}
```

运行核对

腐肉 挥击
小白 射箭

02 动态分派选择实际方法

Java 对实例方法使用动态分派。编译器先确认 `Enemy` 合同里有 `attack`，运行时再执行实际子类的重写版本。

引用找到对象，对象提供重写行为

Enemy enemy



Skeleton 对象

attack() 返回射箭描述

多态不会随机挑方法。引用指向哪个实际对象，就执行那个对象的重写版本。

L22Example02.java • 完整独立示例

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " 普通攻击"; }
}
class Skeleton extends Enemy {
    public String reload() { return name + " 补充箭矢"; }
    Skeleton(String name) { super(name); }
    @Override public String attack() { return name + " 射箭"; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " 挥击"; }
}
public class L22Example02 {
    public static void main(String[] args) {
        Enemy e = new Skeleton("小白");
        System.out.println(e.attack());
        e = new Zombie("腐肉");
        System.out.println(e.attack());
    }
}
```

运行核对

小白 射箭
腐肉 挥击

03 父类数组装下混合波次

数组元素类型写 Enemy，就能把所有 Enemy 子类放在同一波次中。

一个 Enemy 数组保存多种子类

Enemy[]

保存同一家族的引用

数组顺序

决定出场顺序

动态分派

决定攻击方式

以后加入 Witch 子类，循环不需要新增 if。只要 Witch 遵守 Enemy 合同并重写 attack，就能加入波次。

L22Example03.java • 第1/2段，顺序接在同一文件

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " 普通攻击"; }
    public int damage() { return 1; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " 挥击"; }
    @Override public int damage() { return 4; }
}
class Skeleton extends Enemy {
    Skeleton(String name) { super(name); }
    public String reload() { return name + " 补充箭矢"; }
    @Override public String attack() { return name + " 射箭"; }
    @Override public int damage() { return 6; }
}
class Spider extends Enemy {
    Spider(String name) { super(name); }
    @Override public String attack() { return name + " 扑咬"; }
    @Override public int damage() { return 5; }
}
public class L22Example03 {
    static void runWave(Enemy[] wave) {
        if (wave == null) return;
        for (Enemy enemy : wave)
            if (enemy != null)
                System.out.println(enemy.attack() + " 伤害=" + enemy.damage());
    }
}
```

03 父类数组装下混合波次（续）

L22Example03.java • 第2/2段，顺序接在同一文件

```
}
static long totalDamage(Energy[] wave) {
    if (wave == null) return 0;
    long total = 0;
    for (Energy enemy : wave)
        if (enemy != null) total += enemy.damage();
    return total;
}
public static void main(String[] args) {
    Energy[] wave = {new Zombie("腐肉"), new Skeleton("小白"),
                     new Spider("蛛蛛")};
    runWave(wave);
}
}
```

运行核对

腐肉 挥击 伤害=4
小白 射箭 伤害=6
蛛蛛 扑咬 伤害=5

04 方法参数使用父类合同

方法若只需要 Enemy 的能力，参数就写 Enemy，不必为每种子类重复一份方法。

一个入口服务整个敌人家族

参数 Enemy

只依赖共同能力

null 元素

本课明确跳过

空数组

总伤害为0

允许 null 就明确跳过或提示；不允许 null 就尽早拒绝。合同越清楚，波次越不容易中途消失。

L22Example04.java • 第1/2段，顺序接在同一文件

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " 普通攻击"; }
    public int damage() { return 1; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " 挥击"; }
    @Override public int damage() { return 4; }
}
class Skeleton extends Enemy {
    Skeleton(String name) { super(name); }
    public String reload() { return name + " 补充箭矢"; }
    @Override public String attack() { return name + " 射箭"; }
    @Override public int damage() { return 6; }
}
class Spider extends Enemy {
    Spider(String name) { super(name); }
    @Override public String attack() { return name + " 扑咬"; }
    @Override public int damage() { return 5; }
}
public class L22Example04 {
    static void runWave(Enemy[] wave) {
        if (wave == null) return;
        for (Enemy enemy : wave)
            if (enemy != null)
                System.out.println(enemy.attack() + " 伤害=" + enemy.damage());
    }
}
```

04 方法参数使用父类合同（续）

L22Example04.java • 第2/2段，顺序接在同一文件

```
}
static long totalDamage(Energy[] wave) {
    if (wave == null) return 0;
    long total = 0;
    for (Energy enemy : wave)
        if (enemy != null) total += enemy.damage();
    return total;
}
public static void main(String[] args) {
    runWave(new Energy[]{null, new Skeleton("小白")});
    runWave(null);
    System.out.println(totalDamage(new Energy[0]));
}
}
```

运行核对

小白 射箭 伤害=6
0

05 父类引用只看合同入口

即使实际对象是 Skeleton，Enemy 引用也只能直接调用 Enemy 中声明的成员。

对象可能会更多 引用只暴露 Enemy 成员

attack / damage

父类合同中已有

reload

仅骷髅特有

编译检查

父类引用不能直接补箭

如果所有敌人都需要 damage，就把它放进父类合同；只有骷髅需要 reload，就不要强塞给所有敌人。

故障实验：声明类型 Enemy 没有 reload；编译器不让你跳过合同。下面故意保留错误，请先预测，再尝试修复。

BadReload.java • 故意错误示例

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " 普通攻击"; }
}
class Skeleton extends Enemy {
    public String reload() { return name + " 补充箭矢"; }
    Skeleton(String name) { super(name); }
    @Override public String attack() { return name + " 射箭"; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " 挥击"; }
}
public class BadReload {
    public static void main(String[] args) {
        Enemy e = new Skeleton("小白");
        System.out.println(e.reload());
    }
}
```

这个故障文件包含全部辅助类。修复时若确实需要骷髅补箭，用下一节的 instanceof 模式变量。

06 向下转型要有限且安全

向下转型把 Enemy 视为更具体的 Skeleton。只有确实需要子类特有能力和成员时才用，而且必须先检查。

先证明实际类型 再访问子类特有成员

instanceof 检查

是否确实是 Skeleton

模式变量 s

匹配成功后可补箭

强制转型

不匹配会抛异常

大量 instanceof 往往说明父类合同没有表达真正需要的行为，或本应使用接口。先改设计，再考虑强转。

L22Example06.java • 完整独立示例

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " 普通攻击"; }
}
class Skeleton extends Enemy {
    public String reload() { return name + " 补充箭矢"; }
    Skeleton(String name) { super(name); }
    @Override public String attack() { return name + " 射箭"; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " 挥击"; }
}
public class L22Example06 {
    public static void main(String[] args) {
        Enemy e = new Skeleton("小白");
        if (e instanceof Skeleton s) System.out.println(s.reload());
        e = new Zombie("腐肉");
        System.out.println(e instanceof Skeleton);
    }
}
```

运行核对

小白 补充箭矢
false

故障实验：类型关系允许编译，但实际 Zombie 不能转成 Skeleton，运行抛 ClassCastException。下面故意保留错误，请先预测，再尝试修复。

06 向下转型要有限且安全（续）

BadCast.java • 故意错误示例

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " 普通攻击"; }
}
class Skeleton extends Enemy {
    public String reload() { return name + " 补充箭矢"; }
    Skeleton(String name) { super(name); }
    @Override public String attack() { return name + " 射箭"; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " 挥击"; }
}
public class BadCast {
    public static void main(String[] args) {
        Enemy e = new Zombie("腐肉");
        Skeleton s = (Skeleton) e;
        System.out.println(s.reload());
    }
}
```

07 让对象代替类型分支

若主程序根据类型判断攻击方式，新增敌人时每个分支都要修改。重写让对象自己负责行为。

行为写进子类 循环只发送命令

加入 Witch

新类实现攻击

旧循环

不增加类型分支

扩展验证

用输出证明兼容

加入 Witch 后，如果旧循环一行不用改就能输出女巫攻击，说明多态真正发挥了作用。

L22Example07.java • 第1/2段，顺序接在同一文件

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " 普通攻击"; }
    public int damage() { return 1; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " 挥击"; }
    @Override public int damage() { return 4; }
}
class Skeleton extends Enemy {
    Skeleton(String name) { super(name); }
    public String reload() { return name + " 补充箭矢"; }
    @Override public String attack() { return name + " 射箭"; }
    @Override public int damage() { return 6; }
}
class Spider extends Enemy {
    Spider(String name) { super(name); }
    @Override public String attack() { return name + " 扑咬"; }
    @Override public int damage() { return 5; }
}
class Witch extends Enemy {
    Witch(String name) { super(name); }
    @Override public String attack() { return name + " 投掷药水"; }
    @Override public int damage() { return 7; }
}
public class L22Example07 {
```

07 让对象代替类型分支（续）

L22Example07.java • 第2/2段，顺序接在同一文件

```
static void runWave(Enemy[] wave) {
    if (wave == null) return;
    for (Enemy enemy : wave)
        if (enemy != null)
            System.out.println(enemy.attack() + " 伤害=" + enemy.damage());
}
static long totalDamage(Enemy[] wave) {
    if (wave == null) return 0;
    long total = 0;
    for (Enemy enemy : wave)
        if (enemy != null) total += enemy.damage();
    return total;
}
public static void main(String[] args) {
    runWave(new Enemy[]{new Witch("紫袍"), new Spider("蛛蛛")});
}
}
```

运行核对

紫袍 投掷药水 伤害=7
蛛蛛 扑咬 伤害=5

08 波次模拟器测试台

波次程序要检查顺序、每种攻击、总伤害和错误输入。

混合对象 空波次和空元素都要验证

顺序测试

按数组逐项出场

边界测试

空数组和 null

总伤害

各对象 damage 相加

先给每个敌人编号，再记录实际输出。多态负责选行为，数组顺序负责出场顺序。

验收对照表

场景	输入	预期
正常	三种敌人	按顺序输出三种攻击
边界	空数组	无输出 总伤害0
错误	数组含null	跳过 不崩溃
扩展	加入Witch	旧循环无需修改
回归	总伤害	等于各对象damage之和

09 挑战 蓝图统一波次

挑战 程序叫 BattleWave。三个子类分别返回攻击描述和伤害，两个通用方法负责播放波次与统计总伤害。

父类数组统一出场 攻击和伤害统计

runWave

播放每种攻击

totalDamage

只汇总数值

Enemy 合同

两方法共享同一入口

先在 Enemy 中声明 attack 和 damage，再写通用循环。循环不需要知道三种子类的名字。

验收条件

- 同一 Enemy 数组保存三种子类。
- runWave 不出现 instanceof。
- 每个对象输出自己的 attack。
- totalDamage 返回15。
- 空波次返回0，null 元素被安全跳过。

10 挑战 实现战斗波次

新建 BattleWave.java, 在 Java 21 控制台运行。

一个循环触发三种实际行为

三种对象

4、6、5点伤害

null 槽位

明确跳过

long 总数

得到15

在不修改 runWave 和 totalDamage 的前提下加入 Witch。能直接工作，才证明调度代码依赖的是父类合同。

BattleWave.java • 第1/2段, 顺序接在同一文件

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " 普通攻击"; }
    public int damage() { return 1; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " 挥击"; }
    @Override public int damage() { return 4; }
}
class Skeleton extends Enemy {
    Skeleton(String name) { super(name); }
    public String reload() { return name + " 补充箭矢"; }
    @Override public String attack() { return name + " 射箭"; }
    @Override public int damage() { return 6; }
}
class Spider extends Enemy {
    Spider(String name) { super(name); }
    @Override public String attack() { return name + " 扑咬"; }
    @Override public int damage() { return 5; }
}
public class BattleWave {
    static void runWave(Enemy[] wave) {
        if (wave == null) return;
        for (Enemy enemy : wave)
            if (enemy != null)
                System.out.println(enemy.attack() + " 伤害=" + enemy.damage());
    }
}
```

10 挑战 实现战斗波次（续）

BattleWave.java • 第2/2段，顺序接在同一文件

```
}
static long totalDamage(Enemy[] wave) {
    if (wave == null) return 0;
    long total = 0;
    for (Enemy enemy : wave)
        if (enemy != null) total += enemy.damage();
    return total;
}

public static void main(String[] args) {
    Enemy[] wave = {new Zombie("腐肉"), new Skeleton("小白"),
                    new Spider("蛛蛛"), null};

    runWave(wave);
    System.out.println("总伤害=" + totalDamage(wave));
    System.out.println("空波次=" + totalDamage(new Enemy[0]));
}
}
```

运行核对

腐肉 挥击 伤害=4
小白 射箭 伤害=6
蛛蛛 扑咬 伤害=5
总伤害=15
空波次=0

总伤害用 long 累加。这个练习只计算各敌人的 damage，不模拟命中率、护甲或冷却；以后扩展时，应同时修改合同与测试。

11 拓展任务与通关复盘

作业必须标出每个变量的声明类型和实际对象类型。

三档任务训练统一指挥

标声明类型

例如 Enemy

标实际类型

例如 Spider

标调用结果

扑咬来自重写方法

第23课把能力拆成合同。蝙蝠和鞘翅玩家不是一家人，却都能遵守 Flyable 接口。

三档作业

- 木镐：用 Enemy 数组统一输出三种攻击。
- 铁镐：给 Enemy 增加 loot 方法，用多态统一结算掉落。
- 钻石镐：加入 Healer，让相同的 takeTurn 合同既能攻击也能治疗，并说明是否应改用接口。

验收对照表

自评	能	还要练
我理解父类引用	☐	☐
我能解释动态分派	☐	☐
我会避免类型分支	☐	☐

自己的验收记录

漂亮的输出是起点，能解释边界才算掌握。把正常、边界与错误各选一个案例，运行前写预期，运行后留下证据。

为判断留下依据

类别	你选择的操作	核对重点
正常	一个常用输入或一次成功动作	结果是否逐行一致
边界	范围端点与相邻值	有没有多一步、少一项
错误	非法数量、缺失对象或坏数据	是否明确拒绝，状态是否保持

- Bug 记录：触发条件、实际现象、原因、最小修复。
- 图示记录：为每条箭头说明起点、终点和含义。
- 改编记录：只改一个规则，同时修改预期和测试。

技术核对：Java SE 21 语言规范与标准库。类、接口、类型与表达式规则见语言规范；集合、Random、Files 与 Stream 的行为见对应 API。
<https://docs.oracle.com/javase/specs/jls/se21/html/index.html>
<https://docs.oracle.com/en/java/javase/21/docs/api/index.html>