

JAVA / LESSON 22 OF 40

One Command, Different Attacks

Use parent class contract to command different objects

This chapter uses JDK 21 and does not require the preview function. | 90-110 minutes

Making Mixed Battle Waves BattleWave

The commander only shouts attack once, zombie swipes, skeleton shoots arrows, and spider bites. Follow the reference to the real object, and you'll see how Java selects the correct method and completes the mixing wave.

Front-end backpack: Enemy family, extends, super and method rewriting in Lesson 21. Let the entire wave of monsters listen to the same command today.

Before you start: It is recommended to complete Lesson 21 first

Read the full lesson and copy the code:

<https://teacher-ma.de/minecraft-java-lesson.php?lesson=22&lang=en>

Long code lines may wrap at the > margin marker. The marker is not part of the code.

The object is found by reference and the object provides overriding behavior

Element	Explanation
#1 · Skeleton object	attack() returns archery description

Variable	Refers to
Enemy enemy	#1

In this lesson

01. Before starting: predict first, then let the program answer
02. The parent class reference accommodates the subclass object
03. Dynamic dispatch selection actual method
04. Mixed waves under parent class number assembly
05. Method parameters use parent class contract
06. The parent class reference only looks at the contract entry
07. Downward transformation should be limited and safe
08. Let objects replace type branches
09. wave simulator test bench
10. Challenge Blueprint Unification Wave
11. Challenges Realizing Combat Waves
12. Expansion tasks and clearance review
13. Own acceptance record

01 / 13

Before starting: predict first, then let the program answer

This chapter uses JDK 21 and does not require the preview function. The story is borrowed from the block world, and the example runs on the ordinary Java console; if the values and rules are course models, they will be clearly stated.

Each common example is a complete independent file, saved according to the file name marked on the box. Place each example in a different folder to avoid duplicate names of auxiliary classes. When the complete project is divided into sections, they are connected sequentially in the same file.

Terminal command: replace the Chinese placeholder with the actual file name and class name

```
javac -encoding UTF-8 ExampleFile.java
java ExampleClass
```

- Prediction: Write the expected output first, or draw out the changes in references and data.
- Verification: Check line by line after running, and then only change one condition.
- Explanation: Explain whether the results come from Java rules, course contracts, or input data.

"Intentional error" is used to recognize bugs and cannot be mixed with ordinary examples. Use a special temporary directory for file experiments; do not use real archives to test coverage and deletion.

02 / 13

The parent class reference accommodates the subclass object

Enemy enemy = new Skeleton(...) The left side shows that the program uses it through the Enemy contract, and the right side determines that the Skeleton is actually created.

The object is found by reference and the object provides overriding behavior

Element	Explanation
#1 · Skeleton object	attack() returns archery description

Variable	Refers to
Enemy enemy	#1

The reference's declared type is Enemy and the runtime object type is Skeleton or Zombie. Both exist at the same time and should not be confused into one concept.

L22Example01.java · Complete independent example

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " Normal attack"; }
}
class Skeleton extends Enemy {
    public String reload() { return name + " Supplementary arrows"; }
    Skeleton(String name) { super(name); }
    @Override public String attack() { return name + " Archery"; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " Swing"; }
}
public class L22Example01 {
    public static void main(String[] args) {
        Enemy a = new Zombie("Carrion");
        Enemy b = new Skeleton("Xiaobai");
        System.out.println(a.attack());
        System.out.println(b.attack());
    }
}
```

Output

```
Carrion Swing
Xiaobai Archery
```

03 / 13

Dynamic dispatch selection actual method

Java uses dynamic dispatch for instance methods. The compiler first confirms that there is an attack in the Enemy contract, and then executes the rewritten version of the actual subclass at runtime.

The object is found by reference and the object provides overriding behavior

Element	Explanation
#1 · Skeleton object	attack() returns archery description

Variable	Refers to
Enemy enemy	#1

Polymorphism does not randomly pick methods. Whichever actual object the reference points to, the overridden version of that object is executed.

L22Example02.java · Complete independent example

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " Normal attack"; }
}
class Skeleton extends Enemy {
    public String reload() { return name + " Supplementary arrows"; }
    Skeleton(String name) { super(name); }
    @Override public String attack() { return name + " Archery"; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " Swing"; }
}
public class L22Example02 {
    public static void main(String[] args) {
        Enemy e = new Skeleton("Xiaobai");
        System.out.println(e.attack());
        e = new Zombie("Carrion");
        System.out.println(e.attack());
    }
}
```

Output

```
Xiaobai Archery
Carrion Swing
```

04 / 13

Mixed waves under parent class number assembly

If the array element type is written as `Enemy`, all `Enemy` subclasses can be placed in the same wave.

An `Enemy` array holds multiple subclasses

Element	Explanation
<code>Enemy[]</code>	Save references of the same family
Array order	Determine the order of appearance
Dynamic dispatch	Decide on attack method

If the `Witch` subclass is added in the future, there is no need to add a new `if` to the loop. As long as the `Witch` abides by the `Enemy` contract and rewrites the `attack`, she can join the wave.

L22Example03.java · Section 1/2, sequentially connected to the same file

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " Normal attack"; }
    public int damage() { return 1; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " Swing"; }
    @Override public int damage() { return 4; }
}
class Skeleton extends Enemy {
    Skeleton(String name) { super(name); }
    public String reload() { return name + " Supplementary arrows"; }
    @Override public String attack() { return name + " Archery"; }
    @Override public int damage() { return 6; }
}
class Spider extends Enemy {
    Spider(String name) { super(name); }
    @Override public String attack() { return name + " Bite"; }
    @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() + " Damage=" + enemy.damage());
    }
}
```

L22Example03.java · Paragraph 2/2, sequentially connected to the same file

```
}
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("Carrion"), new Skeleton("Xiaobai"),
                    new Spider("Spider")};
    runWave(wave);
}
}
```

Output

```
Carrion Swing Damage=4
Xiaobai Archery Damage=6
Spider Bite Damage=5
```

05 / 13

Method parameters use parent class contract

If the method only requires the ability of `Enemy`, the parameters will be written as `Enemy`, and there is no need to repeat a method for each subclass.

One entrance serves the entire enemy family

Element	Explanation
Parameter <code>Enemy</code>	Rely only on common capabilities
null element	This lesson is explicitly skipped
Empty array	Total damage is 0

If null is allowed, skip or prompt explicitly; if null is not allowed, reject as early as possible. The clearer the contract, the less likely the wave will disappear midway.

L22Example04.java · Section 1/2, sequentially connected to the same file

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " Normal attack"; }
    public int damage() { return 1; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " Swing"; }
    @Override public int damage() { return 4; }
}
class Skeleton extends Enemy {
    Skeleton(String name) { super(name); }
    public String reload() { return name + " Supplementary arrows"; }
    @Override public String attack() { return name + " Archery"; }
    @Override public int damage() { return 6; }
}
class Spider extends Enemy {
    Spider(String name) { super(name); }
    @Override public String attack() { return name + " Bite"; }
    @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() + " Damage=" + enemy.damage());
    }
}
```

L22Example04.java · Paragraph 2/2, sequentially connected to the same file

```
    }
    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[]{null, new Skeleton("Xiaobai")});
        runWave(null);
        System.out.println(totalDamage(new Enemy[0]));
    }
}
```

Output

```
Xiaobai Archery Damage=6
0
```

06 / 13

The parent class reference only looks at the contract entry

Even if the actual object is a Skeleton, an Enemy reference can only directly call members declared in the Enemy.

Objects may have more references exposing only Enemy members

Element	Explanation
attack / damage	Already in the parent class contract
reload	Only unique to skeletons
Compilation check	Parent class reference cannot directly add arrows

If all enemies need damage, put it into the parent contract; only skeletons need reload, so don't force it on all enemies.

Failure experiment: declaring type Enemy without reload; the compiler won't let you skip the contract. Errors are intentionally left below, please anticipate them first and then try to fix them.

BadReload.java · Intentional error example

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " Normal attack"; }
}
class Skeleton extends Enemy {
    public String reload() { return name + " Supplementary arrows"; }
    Skeleton(String name) { super(name); }
    @Override public String attack() { return name + " Archery"; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " Swing"; }
}
public class BadReload {
    public static void main(String[] args) {
        Enemy e = new Skeleton("Xiaobai");
        System.out.println(e.reload());
    }
}
```

This fault file contains all auxiliary classes. If you really need the skeleton to replenish arrows during repair, use the instanceof pattern variable in the next section.

07 / 13

Downward transformation should be limited and safe

Downcast treats Enemy as a more specific Skeleton. Use it only if you really need the subclass-specific capabilities, and you must check them first.

First prove the actual type and then access the subclass-specific members

Element	Explanation
instanceof check	Is it indeed Skeleton?
Mode variable s	Arrows can be replenished after successful matching
Forced transformation	If there is no match, an exception will be thrown.

A large number of instanceof often indicate that the parent class contract does not express the truly required behavior, or that the interface should be used. Change the design first, then consider forced transfer.

L22Example06.java · Complete independent example

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " Normal attack"; }
}
class Skeleton extends Enemy {
    public String reload() { return name + " Supplementary arrows"; }
    Skeleton(String name) { super(name); }
    @Override public String attack() { return name + " Archery"; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " Swing"; }
}
public class L22Example06 {
    public static void main(String[] args) {
        Enemy e = new Skeleton("Xiaobai");
        if (e instanceof Skeleton s) System.out.println(s.reload());
        e = new Zombie("Carrion");
        System.out.println(e instanceof Skeleton);
    }
}
```

Output

```
Xiaobai Supplementary arrows
false
```

Fault experiment: The type relationship allows compilation, but the actual Zombie cannot be converted into Skeleton, and ClassCastException is thrown when running. Errors are intentionally left below, please anticipate them first and then try to fix them.

BadCast.java · Intentional error example

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " Normal attack"; }
}
class Skeleton extends Enemy {
    public String reload() { return name + " Supplementary arrows"; }
    Skeleton(String name) { super(name); }
    @Override public String attack() { return name + " Archery"; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " Swing"; }
}
public class BadCast {
    public static void main(String[] args) {
        Enemy e = new Zombie("Carrion");
        Skeleton s = (Skeleton) e;
        System.out.println(s.reload());
    }
}
```

08 / 13

Let objects replace type branches

If the main program determines the attack method according to the type, each branch will be modified when adding new enemies. Overriding makes the object responsible for its own behavior.

The behavior is written into the subclass and the loop only sends commands

Element	Explanation
Join Witch	New class implementation attack
Old cycle	No type branch added
Extended verification	Prove compatibility with output

After adding Witch, if the old loop can output Witch attack without changing one line, it shows that polymorphism really plays a role.

L22Example07.java · Section 1/2, sequentially connected to the same file

```
class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " Normal attack"; }
    public int damage() { return 1; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " Swing"; }
    @Override public int damage() { return 4; }
}
class Skeleton extends Enemy {
    Skeleton(String name) { super(name); }
    public String reload() { return name + " Supplementary arrows"; }
    @Override public String attack() { return name + " Archery"; }
    @Override public int damage() { return 6; }
}
class Spider extends Enemy {
    Spider(String name) { super(name); }
    @Override public String attack() { return name + " Bite"; }
    @Override public int damage() { return 5; }
}
class Witch extends Enemy {
    Witch(String name) { super(name); }
    @Override public String attack() { return name + " Throw potion"; }
    @Override public int damage() { return 7; }
}
public class L22Example07 {
```

L22Example07.java · Paragraph 2/2, sequentially connected to the same file

```
    static void runWave(Enemy[] wave) {
        if (wave == null) return;
        for (Enemy enemy : wave)
            if (enemy != null)
                System.out.println(enemy.attack() + " Damage=" + 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("Purple robe"), new Spider("Spider")});
    }
}
```

Output

```
Purple robe Throw potion Damage=7
Spider Bite Damage=5
```

09 / 13

wave simulator test bench

The wave program checks the order, each attack, total damage, and incorrect input.

Mixed objects, empty waves and empty elements must be verified

Element	Explanation
Sequential testing	Appearance item by item in array
Border testing	Empty arrays and null
Total damage	Add the damage of each object

Number each enemy first, and then record the actual output. Polymorphism is responsible for the selection behavior, and the array order is responsible for the order of appearance.

Acceptance comparison table

Scene	Enter	Expected
Normal	Three kinds of enemies	Output three attacks in order
Border	Empty array	No output, total damage 0
Error	Array contains null	Skip without crashing
Extension	Join Witch	The old cycle does not need to be modified
Return	Total damage	Equal to the sum of damage of each object

10 / 13

Challenge Blueprint Unification Wave

The challenge program is called BattleWave. The three subclasses return attack description and damage respectively, and the two general methods are responsible for playing waves and counting total damage.

Parent class array unified appearance attack and damage statistics

Element	Explanation
runWave	Play each attack
totalDamage	Only summarize numerical values
Enemy Contract	Two methods share the same entrance

First declare attack and damage in Enemy, and then write the general loop. The loop does not need to know the names of the three subclasses.

Acceptance conditions

- The same Enemy array holds three subclasses.
- instanceof does not appear in runWave.
- Each object outputs its own attack.
- totalDamage returns 15.
- Empty waves return 0, and null elements are safely skipped.

11 / 13

Challenges Realizing Combat Waves

Create a new BattleWave.java and run it in the Java 21 console.

A loop triggers three actual behaviors

Element	Explanation
Three kinds of objects	4, 6, 5 points of damage
null slot	Explicitly skip
long total number	Get 15

Add Witch without modifying runWave and totalDamage. It can work directly, which proves that the scheduling code depends on the parent class contract.

BattleWave.java · Section 1/2, sequentially connected to the same file

```

class Enemy {
    protected final String name;
    Enemy(String name) { this.name = name; }
    public String attack() { return name + " Normal attack"; }
    public int damage() { return 1; }
}
class Zombie extends Enemy {
    Zombie(String name) { super(name); }
    @Override public String attack() { return name + " Swing"; }
    @Override public int damage() { return 4; }
}
class Skeleton extends Enemy {
    Skeleton(String name) { super(name); }
    public String reload() { return name + " Supplementary arrows"; }
    @Override public String attack() { return name + " Archery"; }
    @Override public int damage() { return 6; }
}
class Spider extends Enemy {
    Spider(String name) { super(name); }
    @Override public String attack() { return name + " Bite"; }
    @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() + " Damage=" + enemy.damage());
    }
}

```

BattleWave.java · Paragraph 2/2, sequentially connected to the same file

```
}
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("Carrion"), new Skeleton("Xiaobai"),
                    new Spider("Spider"), null};
    runWave(wave);
    System.out.println("Total damage=" + totalDamage(wave));
    System.out.println("Empty wave times=" + totalDamage(new Enemy[0]));
}
}
```

Output

```
Carrion Swing Damage=4
Xiaobai Archery Damage=6
Spider Bite Damage=5
Total damage=15
Empty wave times=0
```

The total damage is accumulated using long. This exercise only calculates the damage of each enemy and does not simulate hit rate, armor or cooldown; when expanding in the future, the contract and testing should be modified at the same time.

12 / 13

Expansion tasks and clearance review

Assignments must indicate the declared type and actual object type of each variable.

Three levels of mission training unified command

Element	Explanation
Standard declaration type	For example Enemy
mark actual type	For example Spider
Standard call result	Bite comes from rewrite method

Lesson 23 breaks down capabilities into contracts. Bat and elytra players are not the same family, but they both comply with the Flyable interface.

Third gear operation

- Wooden pickaxe: Use the Enemy array to uniformly output three attacks.
- Iron Pickaxe: Add loot method to Enemy, and use polymorphic unified settlement to drop.
- Diamond Pickaxe: Added Healer to allow the same takeTurn contract to attack as well as heal, and indicate whether an interface should be used instead.

Acceptance comparison table

Self-evaluation	Can	Still need to practice
I understand the parent class reference	☐	☐
I can explain dynamic dispatch	☐	☐
I would avoid type branches	☐	☐

13 / 13

Own acceptance record

Beautiful output is the starting point, and being able to explain boundaries is considered mastery. Select one case each from normal, borderline and error, write down your expectations before running, and leave evidence after running.

Leave a basis for judgment

Category	The operation you selected	Check key points
Normal	A common input or a successful action	Is the result consistent line by line?
Border	Range endpoints and adjacent values	Is there one more step or one less thing?
Error	Illegal quantity, missing object or bad data	Whether it is explicitly rejected and whether the status is maintained

- Bug records: triggering conditions, actual phenomena, causes, and minimum repairs.
- Illustrated record: describe the starting point, end point and meaning of each arrow.

- Adaptation record: only change one rule, and modify expectations and tests at the same time.

Technical check: Java SE 21 language specification and standard library. See the language specification for class, interface, type and expression rules; see the corresponding API for the behavior of collections, Random, Files and Stream.

<https://docs.oracle.com/javase/specs/jls/se21/html/index.html>

<https://docs.oracle.com/en/java/javase/21/docs/api/index.html>