

JAVA / LESSON 17 OF 40

Design Mobs with Classes

Class fields and instance methods

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

Making biological arena

Can distinguish categories and objects; can use fields to save the status of each creature; can write instance methods to let objects act; can create multiple Mobs that do not interfere with each other and complete arena testing.

Prerequisites: variables, methods, parameters, return values and scope. Today, move the scattered data and actions into the same blueprint.

Before you start: It is recommended to complete Lessons 12-14 first

Read the full lesson and copy the code:

<https://teacher-ma.de/minecraft-java-lesson.php?lesson=17&lang=en>

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

Same blueprint, two independent objects

Element	Explanation
#1 · Zombie Object	health = 20
#2 · Wolf object	health = 16

Variable	Refers to
zombie	#1
wolf	#2

In this lesson

01. Before you begin: Put the examples in the right place
02. The blueprint is not the creature itself
03. Four-block structure of class 02
04. field saves object status
05. Instance method operation related status
06. Multiple objects do not interfere with each other
07. Method parameters and return values
08. Design Mob's responsibility map
09. Class and Object Fault Clinic
10. Challenge Blueprint Mob Arena
11. Challenges Implementing Mob Arena
12. Expansion tasks and clearance review
13. Own acceptance record

01 / 13

Before you begin: Put the examples in the right place

This chapter uses JDK 21 and does not require the preview function. A complete program is given for each common example, saved with annotated file names. Multiple examples are placed in their own folders to avoid auxiliary classes with the same name.

Terminal command: Replace the Chinese placeholder with the actual name

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

Class names are case-sensitive. If the complete challenge code is divided into multiple sections, please connect them in the same file in order. Do not create separate classes for the continued sections. Run verification only lists the program output; additional input sequences are provided where input is required.

Learning action: first write down your prediction, compare it after running, and then change only one condition. Experiments marked "Intentional Bug" are used to identify bugs; they are clearly distinguished from ready-to-run examples.

02 / 13

The blueprint is not the creature itself

If each zombie was represented by a set of scattered variables, the code would be as messy as a village raid. Classes put related data and actions into a new type.

class is like a workbench recipe, the object is the finished product

Element	Explanation
class	Define common structures
new	Create specific objects
Variables	Save object reference
Status	Instance fields are saved separately

L17Example01.java · Complete independent example

```
class Wolf { String name; int health; }
public class L17Example01 {
    public static void main(String[] args) {
        Wolf wolf = new Wolf();
        wolf.name = "Xiao Hui";
        wolf.health = 16;
        System.out.println(wolf.name);
    }
}
```

Output

Xiao Hui

Four questions

- What common structures do classes define?
- What does new create?
- What does the object variable store?
- Will the fields of the two objects be automatically shared?

Reading: class Wolf is the blueprint; new Wolf() creates a specific wolf; wolf is the variable to find the wolf.

03 / 13

Four-block structure of class 02

Class names are usually named using camel case. Fields are written inside the class body and outside the method, and instance methods can directly access fields of the same object.

The class body encloses fields and methods in curly braces

Element	Explanation
class Mob	Declare new type
Field	Name and life
Example methods	Operate the current object
Curly braces	The class body contains members

L17Example02.java · Complete independent example

```
class Mob {
    String name;
    int health;
    void describe() { System.out.println(name + " Life=" + health); }
}
public class L17Example02 {
    public static void main(String[] args) {
        Mob mob = new Mob();
        mob.name = "Small stiffness";
        mob.health = 20;
        mob.describe();
    }
}
```

Output

```
Small stiffness Life=20
```

Location map: The field belongs to the object state, do not write it into a certain method; the method belongs to the class, do not declare it in another method.

04 / 13

field saves object status

Fields describe what the object looks like now. Use the period notation to select fields of a specific object.

Each object has its own state box

Element	Explanation
zombie	One's own name and life
Wolf	Another name and life
Same blueprint	Same field structure
new twice	Obtain independent object

L17Example03.java · Complete independent example

```
class Mob { String name; int health; }
public class L17Example03 {
    public static void main(String[] args) {
        Mob zombie = new Mob();
        zombie.name = "Small stiffness"; zombie.health = 20;
        Mob wolf = new Mob();
        wolf.name = "Xiao Hui"; wolf.health = 16;
        System.out.println(zombie.health + wolf.health);
    }
}
```

Output

36

Independent backpack: Modifying wolf.health will not change zombie.health. They share blueprints, not field data.

05 / 13

Instance method operation related status

Putting the biological rules into the biological class, the calling site does not need to manually modify health everywhere.

The object allows its own method to modify its own fields

Element	Explanation
takeDamage	Only accept positive damage
health	Deduct blood but not lower than 0
Instance call	Select the object on the left side of the dot number
Next step	Package restrictions are modified directly

L17Example04.java · Complete independent example

```
class Mob {
    String name;
    int health;
    void takeDamage(int n) {
        if (n > 0) health = Math.max(0, health - n);
    }
    int heal(int n) {
        if (n <= 0) return 0;
        int gained = Math.min(n, 20 - health);
        health += gained;
        return gained;
    }
    boolean isAlive() { return health > 0; }
    void describe() { System.out.println(name + " Life=" + health); }
}

public class L17Example04 {
    public static void main(String[] args) {
        Mob zombie = new Mob(); zombie.health = 20;
        Mob wolf = new Mob(); wolf.health = 16;
        zombie.takeDamage(7);
        wolf.takeDamage(3);
        System.out.println(zombie.health + "," + wolf.health);
    }
}
```

Output

```
13,13
```

Responsibility: the legal rules of health are maintained by the Mob method. The caller only says "received 7 points of damage" and is not responsible for manually clipping to 0.

06 / 13

Multiple objects do not interfere with each other

Create three Mobs, and then injure and treat them respectively. Observe that the same method produces different results on different objects.

The same blueprint generates an object army

Element	Explanation
Object a	Received 5 points of damage
Object b	No harm done
Object c	Life dropped to 0
Conclusion	A call only operates the target object

L17Example05.java · Complete independent example

```
class Mob {
    String name;
    int health;
    void takeDamage(int n) {
        if (n > 0) health = Math.max(0, health - n);
    }
    int heal(int n) {
        if (n <= 0) return 0;
        int gained = Math.min(n, 20 - health);
        health += gained;
        return gained;
    }
    boolean isAlive() { return health > 0; }
    void describe() { System.out.println(name + " Life=" + health); }
}

public class L17Example05 {
    public static void main(String[] args) {
        Mob a = new Mob(); a.health = 20;
        Mob b = new Mob(); b.health = 20;
        Mob c = new Mob(); c.health = 16;
        a.takeDamage(5);
        c.takeDamage(20);
        System.out.println(a.health + "," + b.health + "," + c.health);
    }
}
```

Output

```
15,20,0
```

Blueprint reuse: a class is only written once, and new can be executed multiple times. The program gets many objects with a consistent structure and independent state.

07 / 13

Method parameters and return values

Instance methods can have parameters and return values just like static methods. The difference is that it is also automatically connected to the current object.

The example method can also collect materials and return results

Element	Explanation
heal parameter	Request increase amount
Actual increase	No more than remaining space
Return value	Tell the caller the actual increase
isAlive	Only answer whether it is currently alive or not

Here the upper limit of life is clearly set to 20. Calculate how much more can be added first, and then add it up to avoid health + n overflow. Lesson 20 will make fields can only be modified by methods; currently writing fields directly may still break the rules.

Query method: isAlive only reads the status and returns the answer; takeDamage will modify the status. The name should let the reader see the difference.

L17Example06.java · Complete independent example

```
class Mob {
    String name;
    int health;
    void takeDamage(int n) {
        if (n > 0) health = Math.max(0, health - n);
    }
    int heal(int n) {
        if (n <= 0) return 0;
        int gained = Math.min(n, 20 - health);
        health += gained;
        return gained;
    }
    boolean isAlive() { return health > 0; }
    void describe() { System.out.println(name + " Life=" + health); }
}

public class L17Example06 {
    public static void main(String[] args) {
        Mob wolf = new Mob(); wolf.health = 16;
        System.out.println(wolf.heal(10));
        System.out.println(wolf.health);
        System.out.println(wolf.isAlive());
    }
}
```

Output

```
4
20
true
```

08 / 13

Design Mob's responsibility map

Good classes are not about stuffing all your code into new curly braces. Fields and methods should revolve around the responsibilities of the same kind of object.

State and behavior should revolve around the same object

Element	Explanation
name	Save object name
health	Save current life
Rule method	Injury, Treatment and Inquiry
Border	Don't let a single creature control the whole world

Acceptance and inspection forms

Members	Type	Responsibility
name	String field	Save name
health	int field	Save current life
takeDamage	Example methods	Deduct blood and keep the lower limit
heal	Example methods	Return blood and hold the upper limit
isAlive	Example methods	Answer whether alive

Don't cross the line: Weather, world time, and all players list do not belong to a single Mob. Don't let a chicken be responsible for the sun going down.

09 / 13

Class and Object Fault Clinic

The most common mistakes are treating class names as objects, directly accessing instance fields in static main, or forgetting to create objects after declaring variables.

static and instance are two different tracks

Element	Explanation
Class name access	Instance field requires object
Only declare	Local reference not yet initialized
Error static	Possible unexpected sharing of state
Check	Start from the left side of new and dot

Fault experiment: The instance field requires a specific object and cannot be directly read and written with the class name. Errors are intentionally left below, please anticipate them first and then try to fix them.

BadInstance.java · Intentional error example

```
class Mob { int health; }
public class BadInstance {
    public static void main(String[] args) { Mob.health = 10; }
}
```

Fault experiment: The local variable mob is not initialized; the declared reference does not create an object. Errors are intentionally left below, please anticipate them first and then try to fix them.

BadObjectInit.java · Intentional error example

```
class Mob { int health; }
public class BadObjectInit {
    public static void main(String[] args) {
        Mob mob;
        mob.health = 20;
    }
}
```

Emergency sequence: first look for new; then look at who is on the left side of the dot number; finally confirm whether the field or method is incorrectly written as static.

10 / 13

Challenge Blueprint Mob Arena

The challenge program is called MobArena. Mob saves the name, life and attack power; attack injures the target, and describe displays the current status.

Two creatures complete the battle in their own way

Element	Explanation
Create	Two independent organisms
Attack	Call target injury method
Survival check	The fallen will no longer participate
Test	Damage is exactly equal to life

Acceptance conditions

- Create two objects, zombie and iron golem.
- Both sides take turns to attack, and the life is not less than 0.
- No more blood will be deducted when attacking a fallen target.
- It can be seen from the output that the states of the two objects do not interfere with each other.

Acceptance and inspection forms

Test	Operation	Expected
Normal	Zombies attack puppets	Puppet life reduced
Border	Damage equals remaining life	Life becomes 0
Error	Negative attack power	Press 0 to process
Status	Modify a life	The other one remains unchanged

Minimum version: Complete describe and takeDamage first, then write attack. The fewer methods, the quieter the first round of debugging.

11 / 13

Challenges Implementing Mob Arena

Create new MobArena.java. The internal Mob class is used for this lesson demonstration to check the output of each round of combat after running it.

Classes bring state and behavior into the same blueprint

Element	Explanation
Status	Name, life, attack power
Behaviour	Injury, attack, description
main	Arrange the order of battle
Result	The lives of the two objects change respectively

Acceptance: Explain why the health of zombie and golem are independent; change the attack power to a negative number for testing; add another spider.

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

```
public class MobArena {
    static class Mob {
        String name;
        int health;
        int attackPower;

        void takeDamage(int amount) {
            health = Math.max(0, health - Math.max(0, amount));
        }
        void attack(Mob target) {
            if (target != null && isAlive() && target.isAlive()) target.takeDamage(attackPower);
        }
        boolean isAlive() { return health > 0; }
        void describe() {
            System.out.println(name + " Life=" + health);
        }
    }

    static Mob create(String name, int health, int power) {
        Mob mob = new Mob();
        mob.name = name;
        mob.health = Math.max(0, health);
        mob.attackPower = Math.max(0, power);
        return mob;
    }

    public static void main(String[] args) {
        Mob zombie = create("Small stiffness", 20, 5);
```

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

```
        Mob golem = create("Puppet.", 30, 12);
        zombie.attack(golem);
        golem.attack(zombie);
        zombie.describe();
        golem.describe();
        golem.attack(zombie);
        zombie.describe();
        golem.describe();
    }
}
```

Output

```
Small stiffness Life=8
Puppet. Life=25
Small stiffness Life=0
Puppet. Life=25
```

Implement writing the auxiliary Mob as a static nested class so that this file can be run directly. static only modifies nested class declarations; health does not have static and is still saved independently by each object. Practice attack power and life are both customized values.

12 / 13

Expansion tasks and clearance review

The job needs to submit a draft class diagram, source code, running results and a fault record.

The third gear task is to expand the biological design institute

Element	Explanation
Wooden Pickaxe	Design two sheep
Iron Pickaxe	Two objects attack in turn
Diamond Pickaxe	Pet Trust and Behavior
Review	Class, object and instance fields

Third gear operation

- Wooden pickaxe: Design the Sheep class, including color, life and cry methods, to create two sheep.
- Iron Pickaxe: Have two Mobs attack each other, recording lives each round.
- Diamond Pickaxe: Designed to tame the Pet class, including trust value, feeding and following behaviors.

Acceptance and inspection forms

Self-evaluation	Can	Still need to practice
I can differentiate between classes and objects	☐	☐
I can write instance fields	☐	☐
I will let the method maintain the state	☐	☐

Next lesson: Lesson 18 studies the reference actually saved by the object variable. The two variable notes may point to the same pet, or they may point to neither.

13 / 13

Own acceptance record

Write expectations before running, and record actual results after running. Normal cases show that the program does things; edge cases show that it knows when to stop; error cases show that it rejects unreasonable requests.

Write the evidence into the record

Category	Input of your choice	Expectation and actual
Normal	A common value or a successful operation	Is it consistent line by line?
Border	Minimum value, maximum value and adjacent value	Is there one less square and one more step?
Error	Illegal quantity, missing object or incorrect syntax	Explicitly reject? Is the status maintained?

- Write a bug as: triggering conditions, phenomena, causes, and minimal repairs.
- Organize a rule worth remembering and explain it in your own words. Don't just copy the definition.
- Change an example, keep a record of the test, and let the results prove your understanding.

Technical verification basis: Java SE 21 language specification, scope and access control (Chapter 6), classes and constructors (Chapter 8), method invocation (Chapter 15).

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