

JAVA / LESSON 16 OF 40

Project: The Inventory Manager

Comprehensive basic grammar project

This chapter uses JDK 21 and does not require the preview function. | 110-120 minutes

Making an interactive backpack administrator

Can assemble variables, conditions, loops, methods and data into a runnable project; can design menu loops and safe inputs; can complete the four functions of viewing, adding, deleting and searching; can use the test form to accept the backpack administrator.

Prerequisites: variables, inputs, conditions, loops, methods and arrays from Lessons 1 to 15. Today is not about learning a new brick, but about building the old brick into a habitable process.

Before you start: It is recommended to complete Lessons 1-15 first

Read the full lesson and copy the code:

<https://teacher-ma.de/minecraft-java-lesson.php?lesson=16&lang=en>

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

Two sets of arrays, one backpack record

grid	Item	Quantity
0	Wood	12
1	Cobblestone	32
2	null	0

In this lesson

01. Before you begin: Put the examples in the right place
02. Project Map and Acceptance Conditions
03. Backpack data model
04. Menu cycle control program
05. Safe reading of integers
06. View backpack
07. Find items
08. Adding and stacking
09. Delete and clear grid
10. Method Division of Labor
11. Three types of tests
12. Project Failure Clinic
13. Challenge Blueprint
14. Challenge Implementation
15. Expansion tasks and stage review
16. Own acceptance record

01 / 16

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 / 16

Project Map and Acceptance Conditions

The knowledge of the first fifteen lessons will be challenged in a team today. The challenge is not the ender dragon, but a console program that keeps asking "please select a menu".

The final program can view, add, delete, and find items, and will not explode in place when encountering incorrect input.

The backpack administrator draws a map before starting work

Element	Explanation
Menu	Repeat receive operation
logarithmic array	Save name and quantity in the same format
Rule method	View, add, delete, search
Acceptance	Three class: normal, boundary and error

Completion standard

- The menu will be displayed repeatedly, select 0 to exit.
- The item name and quantity are saved using the same index.
- Both addition and deletion check the quantity and knapsack bounds.
- Prepare at least three types of tests: normal, boundary, and error.

Project discipline: Run every time a function is completed. Don't write all four functions at the same time and hope the compiler is in a good mood.

03 / 16

Backpack data model

A grid should store the item name and quantity. We haven't learned about custom item classes at this stage, so we first use two equal-length arrays to pair them.

The same index is a pair of data

Index	items[i]	counts[i]
0	Wood	12
1	Cobblestone	32
2	null	0

L16Example02.java · Complete independent example

```
public class L16Example02 {  
    public static void main(String[] args) {  
        String[] items = {"Wood", "Cobblestone", null};  
        int[] counts = {12, 32, 0};  
        int slot = 1;  
        System.out.println(items[slot] + " x" + counts[slot]);  
    }  
}
```

Output

Cobblestone x32

Pairing rules: items[i] and counts[i] must be modified together. If you only change the name but not the quantity, a supernatural box of "Air x64" will appear.

04 / 16

Menu cycle control program

Menu-type programs usually do not know how many times the player will operate, so they use while. Each round displays the menu, reads the selection, executes the function, and returns to the entrance.

Menu loop: 0 is the exit

Step	Action
1 · Once	running = true
2 · Check	running is true
3 · If true	Display and process
4 · Update; return to step 2	Enter the next round

Exit the loop when the condition is false.

The menu is displayed in each round; the exit setting is running = false, and the input is completed and the user exits safely.

L16Example03.java · Complete independent example

```
public class L16Example03 {
    public static void main(String[] args) {
        java.util.Scanner scanner = new java.util.Scanner(System.in);
        boolean running = true;
        while (running && scanner.hasNextLine()) {
            String choice = scanner.nextLine().trim();
            switch (choice) {
                case "1" -> System.out.println("Open backpack");
                case "0" -> running = false;
                default -> System.out.println("This button does not exist");
            }
        }
        System.out.println("Exited");
    }
}
```

Output

```
Open backpack
This button does not exist
Exited
```

Exit key: The loop condition must be changed to exit the branch. Just print "goodbye" without changing running, and the menu will politely invite you back.

05 / 16

Safe reading of integers

Scanner's `nextLine` first gets the text and then tries to convert it to an integer. When entering "diamond", the program should give a prompt instead of kicking the player out of the world with an exception.

Enter to pass the checkpoint before entering the backpack

Element	Explanation
Text	Read and remove edge margins
<code>parseInt</code>	Try to convert to int
Positive integer	Accept only if successful and greater than 0
Illegal	-1 Report failure

L16Example04.java · Complete independent example

```
public class L16Example04 {
    static int positive(String text) {
        try {
            int n = Integer.parseInt(text.trim());
            return n > 0 ? n : -1;
        } catch (NumberFormatException ex) { return -1; }
    }
    public static void main(String[] args) {
        System.out.println(positive(" 12 "));
        System.out.println(positive("diamond"));
        System.out.println(positive("0"));
        System.out.println(positive("999999999999"));
    }
}
```

Output

```
12
-1
-1
-1
```

Input boundaries: Empty text, 0, negative numbers, and letters are all incorrect inputs. Block them first and then let the backpack method work.

06 / 16

View backpack

The viewing function requires printing of the index, so ordinary for is more appropriate than enhanced for. Spaces are uniformly displayed as "empty spaces".

Ordinary for gets the grid number and content at the same time

Element	Explanation
There are items	Print name and quantity
Vacancy	Uniform display of vacancies
Index	Display 0 to length minus 1
Array	Must be of equal length and legal pairing

L16Example05.java · Complete independent example

```
public class L16Example05 {  
    static void show(String[] items, int[] counts) {  
        for (int i = 0; i < items.length; i++) {  
            String text = items[i] == null ? "Vacancy" : items[i] + " x" + counts[i];  
            System.out.println(i + ": " + text);  
        }  
    }  
    public static void main(String[] args) {  
        String[] items = {"Wood", null};  
        int[] counts = {12, 0};  
        show(items, counts);  
    }  
}
```

Output

```
0: Wood x12  
1: Vacancy
```

Acceptance: A backpack with a length of 5 must output a total of five lines from 0 to 4. To leave out one cell, usually the loop boundary is written as length - 1.

07 / 16

Find items

The search method returns the index, allowing the add, delete and query functions to be reused. Target name checks for null first.

When the search loop encounters the target, it returns to the index

Element	Explanation
target	Exclude null first
equals	Compare by text content
Found	Return to the first index
Not found	Return -1

L16Example06.java · Complete independent example

```
public class L16Example06 {
    static final int MAX_STACK = 64;
    static int find(String[] items, String target) {
        if (target == null) return -1;
        for (int i = 0; i < items.length; i++)
            if (target.equals(items[i])) return i;
        return -1;
    }
    public static void main(String[] args) {
        String[] items = {"Wood", "Cobblestone", null};
        System.out.println(find(items, "Cobblestone"));
        System.out.println(find(items, "diamond"));
        System.out.println(find(items, null));
    }
}
```

Output

```
1
-1
-1
```

Safe comparison: use `target.equals(items[i])`. Even if the backpack cell is null, a `NullPointerException` will not be called.

08 / 16

Adding and stacking

There are two routes for adding items: when the item already exists in the backpack, increase the quantity; when it does not exist, put it into the first space. Maximum 64 per grid.

First find similar items and then find spaces

Element	Explanation
Already exists	Use existing grid
New items	Find the first space
remainder	First check 64 minus the current quantity
Success	Synchronously write name and quantity

L16Example07.java · Complete independent example

```
public class L16Example07 {
    static final int MAX_STACK = 64;
    static int find(String[] items, String target) {
        if (target == null) return -1;
        for (int i = 0; i < items.length; i++)
            if (target.equals(items[i])) return i;
        return -1;
    }
    static int empty(String[] items) {
        for (int i = 0; i < items.length; i++)
            if (items[i] == null) return i;
        return -1;
    }
    static boolean add(String[] items, int[] counts, String item, int n) {
        if (item == null || item.isBlank() || n <= 0) return false;
        int slot = find(items, item);
        if (slot < 0) slot = empty(items);
        if (slot < 0 || n > MAX_STACK - counts[slot]) return false;
        items[slot] = item;
        counts[slot] += n;
        return true;
    }
    public static void main(String[] args) {
        String[] items = {"Wood", null};
        int[] counts = {12, 0};
        System.out.println(add(items, counts, "Wood", 5));
        System.out.println(add(items, counts, "Wood", Integer.MAX_VALUE));
        System.out.println(counts[0]);
    }
}
```

Output

```
true  
false  
17
```

Rules: Each item only occupies one grid, with a maximum of 64 items per grid. Use $n > 64 - \text{counts}[\text{slot}]$ to check the margin first to avoid $\text{counts}[\text{slot}] + n$ overflowing first and then escaping the check. Automatically dividing into multiple piles is an extension task.

Calculate first and then change: first confirm that the new number does not exceed 64, and then write it back to the array. Don't stuff it in first and then find that the lid won't close.

09 / 16

Delete and clear grid

The deletion method first finds the item, and then confirms that the requested quantity is legal. When reduced to 0, the item name must be cleared at the same time.

When the quantity returns to zero, the name and quantity are cleared together.

Element	Explanation
Search	Reject if not found
Quantity	Greater than 0 and not exceeding inventory
Deduction	Modify after passing the inspection
Clear	When returning to zero, the name also becomes null.

No materials owed: Inventory cannot become negative. When 13 pieces of wood are deleted and there are only 12 in the backpack, the method should return false and the data will remain unchanged.

L16Example08.java · Complete independent example

```
public class L16Example08 {
    static final int MAX_STACK = 64;
    static int find(String[] items, String target) {
        if (target == null) return -1;
        for (int i = 0; i < items.length; i++)
            if (target.equals(items[i])) return i;
        return -1;
    }
    static boolean remove(String[] items, int[] counts, String item, int n) {
        int slot = find(items, item);
        if (slot < 0 || n <= 0 || n > counts[slot]) return false;
        counts[slot] -= n;
        if (counts[slot] == 0) items[slot] = null;
        return true;
    }
    public static void main(String[] args) {
        String[] items = {"Wood"};
        int[] counts = {12};
        System.out.println(remove(items, counts, "Wood", 13));
        System.out.println(counts[0]);
        System.out.println(remove(items, counts, "Wood", 12));
        System.out.println(items[0] + ":" + counts[0]);
    }
}
```

Output

```
false
12
true
null:0
```

10 / 16

Method Division of Labor

main is responsible for menus and processes, and functional methods are responsible for data rules. In this way, when an error occurs, it can be quickly determined which "villager" has delivered the wrong goods.

main works as a dispatcher and method works as a professional villager

Element	Explanation
Display	show is responsible for output
Search	find returns index
Modification	add and remove maintenance rules
Enter	main reads and arranges operations

main only selects functions; the reading process is responsible for collecting names and quantities; add/remove only maintains arrays. The complete menu code in this chapter's challenge implementation directly calls the real function without leaving undefined addFlow.

Acceptance and inspection forms

Method	Enter	Return or effect
show	Two arrays	Print the status of each grid
find	Item array, name	Index or -1
add	Array, name, quantity	Success or failure
remove	Array, name, quantity	Success or failure
positive	A line of text	Positive integer or -1

Single responsibility: Method names are like toolbox labels. When you see addFlow, you should know that it handles the adding process, not modifying the weather.

11 / 16

Three types of tests

The project cannot be cleared just by "running it once". Prepare at least one normal test, one boundary test and one error test for each feature.

Test Bench Illustrates Normal Boundaries and Errors

Element	Explanation
Normal	Function can be completed
Border	Just full, just empty
Error	Illegal quantity and unknown command
Return	Rerun the original test after repair

Acceptance and inspection forms

Function	Enter	Expected
Add	Wood 5	The quantity increases by 5
Add	Cobblestone 40 to 32 already	Reject more than 64
Delete	Wood 0	Reject data unchanged
Search	Diamonds that do not exist	Return -1
Menu	x	Prompt no button

Regression testing: fixing a bug and rerunning old tests. Creepers will sometimes come in through another door while you're patching up the wall.

12 / 16

Project Failure Clinic

When the project becomes larger, don't make random changes based on your feelings. Record the input and phenomenon first, then locate the method responsible for the rule.

From phenomenon to responsible method

Element	Explanation
Infinite loop	Exit unchanged condition
Wrong pairing	Name and quantity are out of sync
Crossing the line	Index is illegal
Excess	If the check fails, the data must not be changed

L16Example11.java · Complete independent example

```
public class L16Example11 {  
    public static void main(String[] args) {  
        String[] items = {"Wood"};  
        int[] counts = {12};  
        counts[0] = 0;  
        System.out.println(items[0] + " x" + counts[0]);  
    }  
}
```

Output

```
Wood x0
```

This is not a compilation error, but it breaks the convention that "vacancies correspond to null and 0". When the quantity is reduced to 0, the name of the item in the same grid must be set to null. The same goes for exiting the menu: printing "goodbye" is not the same as modifying running.

Emergency sequence: reproduce first; print choice, slot, amount; confirm the array length; finally make minimal modifications and rerun the test.

13 / 16

Challenge Blueprint

Challenge The program is called InventoryManager. The full version includes a console menu with repeatable operations, with real implementations of viewing, adding, deleting, and searching. Complete the data method first, and then access the menu.

Four core functions of backpack administrator

Element	Explanation
Realize first	View and search
Reimplementation	Add and delete
Take menu	Command calls real function
Acceptance	The two arrays remain unchanged on failure

Acceptance sequence

- Display initial backpack.
- Add 5 pieces of wood and confirm the quantity.
- Add 3 diamonds and confirm that they occupy the space.
- Delete all cobblestones and confirm that the grid is clear.
- Attempts to add that result in a quantity exceeding 64 must fail.

Acceptance and inspection forms

Rules	Legal range	Failure performance
Index	0 to length-1	Do not access array
Quantity	Greater than 0	Return false
Stacking	Maximum 64	Data remains unchanged
Slot	At least one null	Return false

Build the skeleton first: first write show and find and run them, then add add, and finally write remove. Only allow one new method to join the team at a time.

14 / 16

Challenge Implementation

Create new `InventoryManager.java`. The program does not rely on the real Minecraft interface and can be compiled and run directly on the Java 21 console.

The complete program connects five basic grammars into a chain

Element	Explanation
Basics	Array and search
Rules	Balance check and clearing
Menu	while and switch
Enter	Exception handling and end check

Acceptance: Add 5 pieces of wood according to the given input, then check the backpack, the number of pieces of wood should be 17. Continue to manually add 3 diamonds, delete 32 cobblestones, and try to add excess; check the return prompt and backpack status each time.

InventoryManager.java · Section 1/3, sequentially connected to the same file

```
public class InventoryManager {
    static final int MAX_STACK = 64;
    static int find(String[] items, String target) {
        if (target == null) return -1;
        for (int i = 0; i < items.length; i++)
            if (target.equals(items[i])) return i;
        return -1;
    }
    static int empty(String[] items) {
        for (int i = 0; i < items.length; i++)
            if (items[i] == null) return i;
        return -1;
    }
    static boolean add(String[] items, int[] counts, String item, int n) {
        if (item == null || item.isBlank() || n <= 0) return false;
        int slot = find(items, item);
        if (slot < 0) slot = empty(items);
        if (slot < 0 || n > MAX_STACK - counts[slot]) return false;
        items[slot] = item;
        counts[slot] += n;
        return true;
    }
    static boolean remove(String[] items, int[] counts, String item, int n) {
        int slot = find(items, item);
        if (slot < 0 || n <= 0 || n > counts[slot]) return false;
        counts[slot] -= n;
        if (counts[slot] == 0) items[slot] = null;
        return true;
    }
}
```

InventoryManager.java · Section 2/3, sequentially connected to the same file

```

    }
    static void show(String[] items, int[] counts) {
        for (int i = 0; i < items.length; i++) {
            String text = items[i] == null ? "Vacancy" : items[i] + " x" + counts[i];
            System.out.println(i + ": " + text);
        }
    }
    static int positive(String text) {
        try {
            int n = Integer.parseInt(text.trim());
            return n > 0 ? n : -1;
        } catch (NumberFormatException ex) { return -1; }
    }
    public static void main(String[] args) {
        String[] items = {"Wood", "Cobblestone", null, null, null};
        int[] counts = {12, 32, 0, 0, 0};
        java.util.Scanner input = new java.util.Scanner(System.in);
        boolean running = true;
        while (running) {
            System.out.println("1 View 2 Add 3 Delete 4 Search 0 Exit");
            if (!input.hasNextLine()) break;
            String choice = input.nextLine().trim();
            switch (choice) {
                case "0" -> running = false;
                case "1" -> show(items, counts);
                case "2", "3" -> {
                    System.out.println("Item name: ");
                    if (!input.hasNextLine()) { running = false; break; }

```

InventoryManager.java · Section 3/3, sequentially connected to the same file

```

                String item = input.nextLine().trim();
                System.out.println("Positive integer quantity: ");
                if (!input.hasNextLine()) { running = false; break; }
                int n = positive(input.nextLine());
                boolean ok = choice.equals("2")
                    ? add(items, counts, item, n)
                    : remove(items, counts, item, n);
                System.out.println(ok ? "Operation successful" : "Operation rejected, backpack
> unchanged");
            }
            case "4" -> {
                System.out.println("Item name: ");
                if (!input.hasNextLine()) { running = false; break; }
                int slot = find(items, input.nextLine().trim());
                System.out.println(slot < 0 ? "Not found" : "Index: " + slot);
            }
            default -> System.out.println("This button does not exist");
        }
    }
    System.out.println("Backpack is closed");
}
}

```

Output

```
1 View 2 Add 3 Delete 4 Search 0 Exit
Item name:
Positive integer quantity:
Operation successful
1 View 2 Add 3 Delete 4 Search 0 Exit
0: Wood x17
1: Cobblestone x32
2: Vacancy
3: Vacancy
4: Vacancy
1 View 2 Add 3 Delete 4 Search 0 Exit
Backpack is closed
```

Run the check using input: 2, wood, 5, 1, 0, each on a separate line. The menu can be operated repeatedly; the input stream will exit when it ends. The method stipulates that the two arrays are non-null, of equal length, and the initial name and quantity pairing is legal.

15 / 16

Expansion tasks and stage review

This is the challenge of the basic grammar stage. Submit source code, running results, test tables and a bug record.

The third-level expansion makes the backpack truly look like a project

Element	Explanation
Wooden Pickaxe	Reproduce basic menu
Iron Pickaxe	Completion test and gap statistics
Diamond Pickaxe	Studying multiple stacks
Review	Record data rules

Third gear operation

- Wooden pickaxe: Press the complete menu to reproduce viewing, adding, and exiting; submit input and output records.
- Iron Pick: Complete the boundary test of deletion and search, add space statistics, and confirm that incorrect input will not change the data.
- Diamond Pickaxe: Allows each item to have multiple stacks, automatically looking for grids less than 64.

Acceptance and inspection forms

Self-evaluation	Can	Still need to practice
I can split project methods	☐	☐
I can maintain an array of pairs	☐	☐
I can write three types of tests	☐	☐

Next lesson: Lesson 17: Enter the Biological Design Institute. We will use classes to pack state and behavior into the same creature blueprint.

16 / 16

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>