

JAVA / LESSON 25 OF 40

A Mob List That Grows

Use ArrayList to edit dynamic waves

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

Make dynamic wave editor DynamicWave

The wave editor can add monsters, change monsters, and delete monsters at any time. A growing list gives you freedom, but also requires you to recheck the index. You'll use testing to prove that failed actions didn't change the team.

Prerequisites: arrays, loops, methods and objects. The array is like a fixed hotbar, and the ArrayList is like an expanding warehouse, but the administrator number still starts from 0.

Before you start: It is recommended to complete Lessons 15 and 21-24 first

Read the full lesson and copy the code:

<https://teacher-ma.de/minecraft-java-lesson.php?lesson=25&lang=en>

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

Both containers store elements of the same type in order

Element	Explanation
Array	The length is fixed after creation
ArrayList	Automatic management of capacity
size()	Current number of elements

In this lesson

01. Before starting: predict first, then let the program answer
02. The array is fixed but the list will expand
03. Generic restriction element type
04. add get and size
05. set and remove by index
06. Delete index or delete object
07. Enhance for reading queue
08. Index loop supports controlled modification
09. removeIf security filtering
10. Empty list and index boundary
11. methods preferentially receive List contracts
12. Three types of test dynamic queues
13. Challenge Blueprint Dynamic Wave Editor
14. Challenges Realizing Dynamic Waves
15. Expansion tasks and unit review
16. Own acceptance record

01 / 16

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 / 16

The array is fixed but the list will expand

The number of monster waves changes constantly while editing. Arrays require a predetermined length, and ArrayList can be added or removed as needed.

Both containers store elements of the same type in order

Element	Explanation
Array	The length is fixed after creation
ArrayList	Automatic management of capacity
size()	Current number of elements

ArrayList automatically manages internal capacity, but is still limited by memory. "Will expand" in the course means that there is no need to manually rebuild a larger array.

Acceptance comparison table

Compare	Array	ArrayList
Length	Fixed after creation	Can change dynamically
Read	items[i]	list.get(i)
Modification	items[i] = x	list.set(i, x)
Add	Need to create another array	list.add(x)
Length query	length	size()

03 / 16

Generic restriction element type

Generics block errors at the compilation stage. `ArrayList<String>` means that the list elements must be Strings.

Angle brackets indicate which items the queue only accepts

Element	Explanation
List<String>	Element types are checked at compile time
ArrayList<>	Create variable implementation
Generic	Cannot add incompatible values

Don't write a bare ArrayList. Lack of generics results in loss of type checking and dangerous coercion when removing elements.

L25Example02.java · Complete independent example

```
import java.util.*;
public class L25Example02 {
    public static void main(String[] args) {
        List<String> wave = new ArrayList<>();
        wave.add("Zombie");
        System.out.println(wave);
    }
}
```

Output

```
[Zombie]
```

Fault experiment: String list cannot add int value. Errors are intentionally left below, please anticipate them first and then try to fix them.

BadGeneric.java · Intentional error example

```
import java.util.*;
public class BadGeneric {
    public static void main(String[] args) {
        List<String> wave = new ArrayList<>();
        wave.add(20);
    }
}
```

04 / 16

add get and size

add puts the element at the end, and size returns the current number of elements. When the list has 3 elements, the legal indexes are 0, 1, and 2.

Add to the end Read with index Use size to query the quantity

Element	Explanation
add(x)	Add at the end
add(i,x)	Insert and move subsequent items
get(i)	Read legal index

After inserting Creeper at position No. 1, the original No. 1 and subsequent elements will move to the right as a whole, and the index will change.

L25Example03.java · Complete independent example

```
import java.util.*;
public class L25Example03 {
    public static void main(String[] args) {
        List<String> wave = new ArrayList<>();
        wave.add("Zombie"); wave.add("Skeleton");
        wave.add(1, "Creeper");
        System.out.println(wave);
        System.out.println(wave.size());
        System.out.println(wave.get(2));
    }
}
```

Output

```
[Zombie, Creeper, Skeleton]
3
Skeleton
```

05 / 16

set and remove by index

set replaces the specified element, and the list size remains unchanged; remove(index) deletes the element, and the size is reduced by 1.

Replacement does not change the size. Deletion will move the following elements to the left.

Element	Explanation
set(i,x)	Replacement, size remains unchanged
remove(i)	Delete, reduce size by 1
Right element	Index moves left

After deleting an element, all indexes on the right side are reduced by 1. If you save an old index, re-verify who it points to before continuing to use it.

L25Example04.java · Complete independent example

```
import java.util.*;
public class L25Example04 {
    public static void main(String[] args) {
        List<String> wave = new ArrayList<>(List.of("Zombie", "Skeleton"));
        System.out.println(wave.set(1, "Spider"));
        System.out.println(wave.remove(0));
        System.out.println(wave);
    }
}
```

Output

```
Skeleton
Zombie
[Spider]
```

06 / 16

Delete index or delete object

`remove(1)` usually means deleting position No. 1, and `remove("Spider")` means deleting the first matching object. Lists of integers require special care.

remove overload selects version according to parameter type

Element	Explanation
<code>remove(int)</code>	Parameter is index
<code>remove(Object)</code>	Delete first matching value
<code>Integer.valueOf</code>	Explicitly select object for deletion

When seeing `remove`, first ask whether the `int` index or an object is passed in. When the generic is `Integer`, both versions may look like numbers.

L25Example05.java · Complete independent example

```
import java.util.*;
public class L25Example05 {
    public static void main(String[] args) {
        List<Integer> ids = new ArrayList<>(List.of(10, 20, 30));
        System.out.println(ids.remove(1));
        System.out.println(ids.remove(Integer.valueOf(10)));
        System.out.println(ids);
    }
}
```

Output

```
20
true
[30]
```

07 / 16

Enhance for reading queue

When you only want to read elements sequentially, enhanced for is most suitable. Direct add or remove during traversal may trigger `ConcurrentModificationException`.

Read-only traversal is concise and clear

Element	Explanation
Enhance for	Suitable for sequential reading
Structural modification	Will make iteration invalid
Controlled deletion	Use <code>removeIf</code> and other solutions

Enhanced to treat the list structure as read-only during for. Use `removeIf`, `Iterator`, or reverse index loop when deletion is required.

L25Example06.java · Complete independent example

```
import java.util.*;
public class L25Example06 {
    public static void main(String[] args) {
        List<String> wave = new ArrayList<>(List.of("Zombie", "Spider"));
        for (String mob : wave) System.out.println(mob);
    }
}
```

Output

```
Zombie
Spider
```

Fault experiment: This set of data throws `ConcurrentModificationException` when continuing to iterate; do not rely on this exception to ensure that every violation can be discovered. Errors are intentionally left below, please anticipate them first and then try to fix them.

BadIteration.java · Intentional error example

```
import java.util.*;
public class BadIteration {
    public static void main(String[] args) {
        List<String> wave = new ArrayList<>(List.of("A", "B", "C"));
        for (String mob : wave) wave.remove(mob);
    }
}
```

08 / 16

Index loop supports controlled modification

Ordinary for is suitable when replacing a specified position. If you delete multiple elements in a forward direction, do not skip the new elements that have just been moved after deletion.

Use ordinary for when position is needed

Element	Explanation
Reverse order cycle	Starting from the last index
Delete current item	Do not change smaller index
No skip items	Adjacent matching can also be deleted

Removing from the end does not change smaller indices that have not been visited yet, so elements are not skipped.

L25Example07.java · Complete independent example

```
import java.util.*;
public class L25Example07 {
    public static void main(String[] args) {
        List<String> wave = new ArrayList<>(List.of("Zombie", "Zombie", "Spider"));
        for (int i = wave.size() - 1; i >= 0; i--)
            if ("Zombie".equals(wave.get(i))) wave.remove(i);
        System.out.println(wave);
    }
}
```

Output

[Spider]

09 / 16

removelf security filtering

When conditionally removing all matches, removelf is more straightforward than a handwritten index loop.

Give the deletion conditions to the list for unified execution

Element	Explanation
removelf	Delete matches conditionally
constant.equals	Safe handling null
Variable list	Necessary prerequisite for deletion

If the list allows null, first check mob != null in the condition, or put the constant to the left of equals.

L25Example08.java · Complete independent example

```
import java.util.*;
public class L25Example08 {
    public static void main(String[] args) {
        List<String> wave = new ArrayList<>();
        wave.add(null); wave.add("Zombie"); wave.add("Spider");
        wave.removeIf("Zombie"::equals);
        System.out.println(wave);
    }
}
```

Output

```
[null, Spider]
```

`equals` for constant strings safely accepts null. `removeIf` modifies the list, requiring the caller to pass in a container that supports removal; the results of `List.of` and `Stream.toList` cannot be used for such modifications.

10 / 16

Empty list and index boundary

An empty list can exist normally, but `get(0)` will throw `IndexOutOfBoundsException`.

When size is 0, there is no element No. 0

Element	Explanation
Empty list	Can exist but cannot <code>get(0)</code>
Legal index	$0 \leq i < \text{size}()$
<code>size()</code>	Not the last index

The legal condition is always `index >= 0 && index < list.size()`. Do not write less than or equal to size.

L25Example09.java · Complete independent example

```
import java.util.*;
public class L25Example09 {
    static String safeGet(List<String> wave, int i) {
        return wave != null && i >= 0 && i < wave.size() ? wave.get(i) : null;
    }
    public static void main(String[] args) {
        List<String> wave = new ArrayList<>();
        System.out.println(safeGet(wave, 0));
        wave.add("Spider");
        System.out.println(safeGet(wave, 0));
        System.out.println(safeGet(wave, 1));
    }
}
```

Output

```
null
Spider
null
```

Acceptance comparison table

size	Legal index	Annotations
0	None	get(0) out of bounds
1	0	Only the first item
3	0 to 2	Finally, size minus 1

11 / 16

methods preferentially receive List contracts

If the method only requires the common operations of List, the parameter type can be written as List<String>, and the caller can still pass in ArrayList.

The caller can choose different list implementations

Element	Explanation
Parameter List	Rely on universal contract
Object ArrayList	Provide variable implementation
Return false	Illegal requests are not modified

The parameter writing List indicates that the method does not rely on the special implementation of ArrayList. New ArrayList can still be used when creating objects.

L25Example10.java · Complete independent example

```
import java.util.*;

public class L25Example10 {
    static boolean addMob(List<String> wave, String mob) {
        if (wave == null || mob == null || mob.isBlank()) return false;
        wave.add(mob.strip());
        return true;
    }

    static boolean replace(List<String> wave, int index, String mob) {
        if (wave == null || index < 0 || index >= wave.size()
            || mob == null || mob.isBlank()) return false;
        wave.set(index, mob.strip());
        return true;
    }

    static int removeAll(List<String> wave, String mob) {
        if (wave == null || mob == null || mob.isBlank()) return 0;
        String target = mob.strip();
        int before = wave.size();
        wave.removeIf(target::equals);
        return before - wave.size();
    }

    static void show(List<String> wave) {
        if (wave == null) { System.out.println("No queue"); return; }
        for (int i = 0; i < wave.size(); i++)
            System.out.println(i + ": " + wave.get(i));
    }

    public static void main(String[] args) {
        List<String> wave = new ArrayList<>();
        System.out.println(addMob(wave, " Spider "));
        System.out.println(replace(wave, 1, "Zombie"));
        System.out.println(wave);
    }
}
```

Output

```
true  
false  
[Spider]
```

12 / 16

Three types of test dynamic queues

Dynamic containers change frequently, and testing must simultaneously check the return value, size, and element order.

Add, replace, delete all verification status

Element	Explanation
Previous snapshot	Copy element sequence
Perform operations	Only test one item at a time
Post comparison	Return value, size, order

Copy a list before the failed operation and compare the contents after the operation. Returning false but the data has changed is still a serious bug.

Acceptance comparison table

Scene	Operation	Expected
Normal	Add Zombie	size increased by 1
Normal	Replace index 1	Size remains unchanged but elements change
Border	Delete the last one	Successful and in correct order
Error	Index equals size	Return false and the data remains unchanged
Error	Add empty name	Return false and the data remains unchanged

13 / 16

Challenge Blueprint Dynamic Wave Editor

The challenge program is called `DynamicWave`. It receives `List<String>` parameter, and the main program uses `ArrayList` to save the waves.

Add, replace, delete and display the composition editing process

Element	Explanation
<code>addMob</code>	Only responsible for adding
<code>replace</code>	Only responsible for replacement
<code>removeAll</code>	Delete all by name

`addMob` only adds, `replace` only replaces, `removeAll` only filters by name, and `show` only displays. It is easier to locate bugs when responsibilities are clear.

Acceptance sequence

- Add Zombie, Skeleton, Zombie in sequence.
- Replace element No. 1 with Spider, and the size is still 3.
- Delete all Zombies, 2 should be deleted.
- Finally, position No. 0 is displayed as Spider.
- Illegal indexes and empty names return false, and the content and order of the list remain unchanged.

14 / 16

Challenges Realizing Dynamic Waves

Create a new `DynamicWave.java`, compile and run using Java 21.

Generic lists are safely edited via methods

Element	Explanation
Add three items	Zombie, Skeleton, Zombie
Replace the middle	Change to Spider
Filter zombies	One Spider left

Change wave to run with an empty list; add empty strings; delete non-existent witches; confirm failed operations without changing the list.

DynamicWave.java · Complete stand-alone example

```
import java.util.*;

public class DynamicWave {
    static boolean addMob(List<String> wave, String mob) {
        if (wave == null || mob == null || mob.isBlank()) return false;
        wave.add(mob.strip());
        return true;
    }

    static boolean replace(List<String> wave, int index, String mob) {
        if (wave == null || index < 0 || index >= wave.size()
            || mob == null || mob.isBlank()) return false;
        wave.set(index, mob.strip());
        return true;
    }

    static int removeAll(List<String> wave, String mob) {
        if (wave == null || mob == null || mob.isBlank()) return 0;
        String target = mob.strip();
        int before = wave.size();
        wave.removeIf(target::equals);
        return before - wave.size();
    }

    static void show(List<String> wave) {
        if (wave == null) { System.out.println("No queue"); return; }
        for (int i = 0; i < wave.size(); i++)
            System.out.println(i + ": " + wave.get(i));
    }

    public static void main(String[] args) {
        List<String> wave = new ArrayList<>();
        addMob(wave, "Zombie"); addMob(wave, "Skeleton");
        addMob(wave, "Zombie");
        System.out.println("Replacement=" + replace(wave, 1, "Spider"));
        System.out.println("Delete=" + removeAll(wave, "Zombie"));
        System.out.println("Reject empty names=" + !addMob(wave, " "));
        show(wave);
    }
}
```

Output

```
Replacement=true
Delete=2
Reject empty names=true
0: Spider
```

The List contract does not promise to allow additions or deletions. This project requires passing in a variable list, such as ArrayList; illegal indexes and empty names will return failure and the data will not be modified.

15 / 16

Expansion tasks and unit review

Lessons 21 to 25 complete inheritance, polymorphism, interfaces, value objects, and dynamic collections. The job submits code, test sheets and a fault record at the same time.

The third-level task continues to expand the data warehouse

Element	Explanation
Type design	Inheritance, interface, value object
Container operations	Generics, indexes, additions and deletions
Delivery evidence	Code and test records

Lesson 26 uses HashMap to manage player scoring. The name is used as the key, the score is used as the value, and the 100th variable is no longer created for the 100th player.

Third gear operation

- Wooden Pickaxe: Use ArrayList to manage task lists, supporting addition, completion and display.
- Iron Pickaxe: Write removeAllByName to delete all creatures with the same name and return the number deleted.
- Diamond Pickaxe: Use Collections.shuffle to randomly shuffle waves and keep a copy of the original order for comparison.

Acceptance comparison table

Self-evaluation	Can	Still need to practice
I would use a generic list	☐	☐
I can safely add and delete elements	☐	☐
I would test list boundaries	☐	☐

16 / 16

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>