

JAVA / LESSON 28 OF 40

Protect Your Program from Errors

Read abnormally, and let failure lead to clear results.

This chapter uses JDK 21 and does not require the preview function. | 100-120 minutes

Make safe backpack administrator SafeInventory

The user inputs "a lot" and the program does not have to end immediately. You'll read exception clues, catch recoverable failures, check all conditions before writing to inventory, and provide actionable prompts.

Prerequisites: methods, conditional judgments, Map and string conversion. Today, when the player inputs a lot of diamonds, the program will not explode on the spot like a creeper.

Before you start: It is recommended to complete Lessons 25-27 first

Read the full lesson and copy the code:

<https://teacher-ma.de/minecraft-java-lesson.php?lesson=28&lang=en>

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

Type message and call location constitute fault clues

Element	Explanation
Exception type	Explain which type of operation failed
Message	Give specific input clues
Source file line number	Locate your own code

In this lesson

01. Before starting: predict first, then let the program answer
02. Exception reporting failed operation
03. try catch handles recoverable errors
04. Capture specific exceptions
05. Ordinary verification does not need to throw exceptions
06. throw reject bad parameters
07. finally and resource cleanup
08. Detected and Unchecked Abnormalities
09. Custom abnormal expression field failure
10. Error messages must be actionable
11. Abnormal paths must also be tested
12. Challenge Blueprint Security Backpack Manager
13. safety backpack program
14. Expansion tasks and clearance review
15. Own acceptance record

01 / 15

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 / 15

Exception reporting failed operation

An exception is not a notification that the programmer has been deducted points, but a fault report sent when it is discovered that the operation cannot be completed normally during runtime.

The stack trace lists the method call path. First look at the exception type and message, and then find the first line pointing to your source file.

Type message and call location constitute fault clues

Element	Explanation
Exception type	Explain which type of operation failed
Message	Give specific input clues
Source file line number	Locate your own code

The error message is a clue. Only keep the exception type, message and line number in your own code to start locating.

Fault experiment: `NumberFormatException` is thrown during runtime; find the stack line pointing to `BadNumber.java`. Errors are intentionally left below, please anticipate them first and then try to fix them.

BadNumber.java · Intentional error example

```
public class BadNumber {  
    public static void main(String[] args) {  
        int amount = Integer.parseInt("Many");  
        System.out.println(amount);  
    }  
}
```

03 / 15

try catch handles recoverable errors

Converting string to integer may fail. If the user can re-enter, catch the `NumberFormatException` and return an understandable result.

Dangerous operations are put into try and corresponding exceptions are handed over to catch

Element	Explanation
try	Only put operations that may fail
catch	Handling recoverable errors
Return -1 on failure	Not to be confused with legal positive numbers

Only operations that may actually fail are placed in try. When the range is too large, it is difficult for catch to tell which step went wrong.

L28Example02.java · Complete independent example

```
public class L28Example02 {  
    static int parseAmount(String text) {  
        if (text == null) return -1;  
        try {  
            int amount = Integer.parseInt(text.strip());  
            return amount > 0 ? amount : -1;  
        } catch (NumberFormatException error) { return -1; }  
    }  
    public static void main(String[] args) {  
        System.out.println(parseAmount("12"));  
        System.out.println(parseAmount("Many"));  
    }  
}
```

Output

```
12  
-1
```

Returning -1 here represents failure, and the legal value is a positive integer. Don't confuse legal 0 with "parsing failure"; if 0 is also legal, you can design the return result object to carry the success flag and value respectively.

04 / 15

Capture specific exceptions

Although catching Exception saves words, it may mislead program errors into user input problems. Prioritize capturing specific types that you can handle explicitly.

Different faults require different repair solutions

Element	Explanation
Specific exception	NumberFormatException first
Parent class exception	IllegalArgumentException follows
Unrecoverable	Keep spreading, don't pretend to be successful

A catch should restore, transform, or supplement the context. If it cannot be handled correctly, let the exception continue to propagate.

L28Example03.java · Complete independent example

```
public class L28Example03 {  
    public static void main(String[] args) {  
        for (String text : new String[]{"Many", "80"}) {  
            try {  
                int amount = Integer.parseInt(text);  
                if (amount < 1 || amount > 64)  
                    throw new IllegalArgumentException(  
                        "The single quantity must be within 1 to 64 between: " + amount);  
                System.out.println("The quantity is legal=" + amount);  
            } catch (NumberFormatException error) {  
                System.out.println("Quantity must be an integer: " + text);  
            } catch (IllegalArgumentException error) {  
                System.out.println(error.getMessage());  
            }  
        }  
    }  
}
```

Output

```
Quantity must be an integer: Many  
The single quantity must be within 1 to 64 between: 80
```

`NumberFormatException` is a subclass of `IllegalArgumentException`, which must be caught more specifically first. If the order is reversed, the subsequent catch cannot be reached and the compiler will report an error.

05 / 15

Ordinary verification does not need to throw exceptions

It is a common business situation for players to enter an empty name, and verification rules can be directly expressed using `if`. Exceptions are used for operations that cannot be completed normally; the choice depends on the calling contract, rather than treating exceptions as ordinary branching loops.

It is expected that you can choose to use `if` to handle the actual failure and then hand over the exception.

Element	Explanation
Common verification	For empty names, use <code>if</code> first
Clear the contract	What is the legal input?
No side effects	Operate only after passing inspection

Use if for situations that often occur in the normal process; exceptions will be considered when the library or operation cannot be completed.

L28Example04.java · Complete independent example

```
public class L28Example04 {  
    static boolean validName(String name) { return name != null && !name.isBlank(); }  
    public static void main(String[] args) {  
        System.out.println(validName(" "));  
        System.out.println(validName(" Alex "));  
    }  
}
```

Output

```
false  
true
```

06 / 15

throw reject bad parameters

If the method requires a single number of 1 to 64, check it at the entrance. throw hands the error to the caller rather than continuing to save bad data.

If the method finds that the contract is broken, it will be reported immediately.

Element	Explanation
Entry Check	Reject if the parameter is illegal
throw	Leave it to the caller for processing
Message	Inclusion range and received value

Just writing "error" doesn't help. The fix will be much faster by writing out the parameter names, received values, and legal ranges.

L28Example05.java · Complete independent example

```
public class L28Example05 {  
    static void checkAmount(int amount) {  
        if (amount < 1 || amount > 64)  
            throw new IllegalArgumentException("The quantity should be 1 to 64: " + amount);  
    }  
    public static void main(String[] args) {  
        try { checkAmount(-1); }  
        catch (IllegalArgumentException e) { System.out.println(e.getMessage()); }  
        checkAmount(64);  
        System.out.println("64 Through");  
    }  
}
```

Output

```
The quantity should be 1 to 64: -1  
64 Through
```

07 / 15

finally and resource cleanup

Finishing is performed along the control path. `finally` is suitable for ordinary closing; resources that need to be closed are given priority by using `try-with-resources` to avoid writing the success and failure branches separately.

When returning normally or throwing an exception, end along the control path

Element	Explanation
<code>finally</code>	Ending on control path
Automatic resource management	<code>try-with-resources</code>
Do not return	Avoid overwriting original exception or result

`Finally` ends on normal and abnormal control paths, but there is no guarantee for process termination and other situations. Do not return in it to avoid obscuring the original exception or return value.

L28Example06.java · Complete independent example

```
import java.util.*;
public class L28Example06 {
    public static void main(String[] args) {
        try { Integer.parseInt("Many"); }
        catch (NumberFormatException e) { System.out.println("Requires integer"); }
        finally { System.out.println("This attempt ends"); }
        try (Scanner scanner = new Scanner("3")) {
            System.out.println(scanner.nextInt());
        }
    }
}
```

Output

```
Requires integer
This attempt ends
3
```

finally executes on the control path of normal return and abnormal exit; there is no guarantee when the JVM is terminated, the process crashes, etc. Don't return in finally. What is closed here is the string Scanner; closing the Scanner that wraps System.in will also close the input stream.

08 / 15

Detected and Unchecked Abnormalities

Checked exceptions are required by the compiler to be caught or declared in the method signature. Unchecked exceptions usually indicate that parameters or program state do not conform to the contract.

Compiler Requirements to Handle Partially Recoverable Failures

Element	Explanation
IOException	Compiler requires processing or declaration
RuntimeException	Unchecked exception
Inspection method	Different does not mean different importance

The two types of exceptions are compiled and checked in different ways, which does not mean that one type is important and the other is not.

L28Example07.java · Complete independent example

```
public class L28Example07 {  
    static void read() throws java.io.IOException {  
        throw new java.io.IOException("Demo read failed");  
    }  
    public static void main(String[] args) {  
        try { read(); }  
        catch (java.io.IOException e) { System.out.println(e.getMessage()); }  
    }  
}
```

Output

```
Demo read failed
```

Acceptance comparison table

Category	Example	Common processing
Detected exception	IOException	catch or throws
Unchecked exception	IllegalArgumentException	Modify parameter contract
Unchecked exception	NumberFormatException	User input can be captured

09 / 15

Custom abnormal expression field failure

When multiple backpack rules may fail, a custom exception lets the caller know explicitly that this is an issue with the inventory domain.

InventoryException indicates that the backpack operation failed

Element	Explanation
InventoryException	Expressing a type of field failure
extends Exception	This is designed as a checked exception
Caller	Explicitly choose how to restore

A custom exception is created only when the caller needs to uniformly distinguish and handle a type of field failure. Single simple parameter errors can be made directly using standard types.

L28Example08.java · Complete independent example

```
class InventoryException extends Exception {
    InventoryException(String message) { super(message); }
}
public class L28Example08 {
    static int withdraw(int stock, int amount) throws InventoryException {
        if (stock < 0 || amount <= 0 || amount > stock)
            throw new InventoryException("Insufficient inventory");
        return stock - amount;
    }
    public static void main(String[] args) {
        try { withdraw(0, 1); }
        catch (InventoryException e) { System.out.println(e.getMessage()); }
    }
}
```

Output

```
Insufficient inventory
```

10 / 15

Error messages must be actionable

Good news allows users to know what happened and how to change it next, and also allows programmers to find corresponding operations.

Write clearly the input operation and legal range.

Element	Explanation
What happened	Add operation failed
Which input	Item name or quantity
Next step	Provide legal scope and correction suggestions

The prompts for students are in clear Chinese; the debugging log can retain the exception type and reason. Don't mix the two into one big stack.

Acceptance comparison table

Message	Problem	Improvement
Error	No context	Quantity must be an integer: many
failed	No operation name	Failed to add diamond: item name is empty
Wrong range	No legal scope	Quantity must be between 1 and 64: 80

11 / 15

Abnormal paths must also be tested

Only testing normal input is equivalent to only coloring the armor. True blast armor is meant to withstand impacts in the wrong path.

Successful capture, throwing and unchanged status must be accepted

Element	Explanation
Normal operation	Increase in quantity
Parsing failed	Chinese quantities cannot be converted to integers
Status comparison	Inventory will not be changed for any failure

Copy the inventory before failure and compare the Map after failure. The prompt is correct but the data has been modified and the level is still not cleared.

L28Example10.java · Section 1/2, sequentially connected to the same file

```
import java.util.*;
public class L28Example10 {
    static void addItem(Map<String, Integer> inventory,
                        String item, int amount) {
        if (inventory == null)
            throw new IllegalArgumentException("Inventory cannot be null");
        if (item == null || item.isBlank())
            throw new IllegalArgumentException("Item name cannot be empty");
        if (amount < 1 || amount > 64)
            throw new IllegalArgumentException("The single quantity must be within 1 to 64 between: "
> + amount);
        String key = item.strip();
        Integer old = inventory.get(key);
        if (inventory.containsKey(key) && (old == null || old < 0))
            throw new IllegalArgumentException("The original inventory record is illegal: " + key);
        long total = (long) (old == null ? 0 : old) + amount;
        if (total > Integer.MAX_VALUE)
            throw new IllegalArgumentException("The cumulative inventory exceeds the upper limit of
> the integer: " + key);
        inventory.put(key, (int) total);
    }
    static String process(Map<String, Integer> inventory,
                          String item, String amountText) {
        if (item == null || item.isBlank()) return "Item name cannot be empty";
        if (amountText == null || amountText.isBlank()) return "Quantity cannot be empty";
        String key = item.strip();
        try {
            int amount = Integer.parseInt(amountText.strip());
            addItem(inventory, key, amount);
            return key + " Added successfully Quantity=" + inventory.get(key);
        } catch (NumberFormatException error) {
            return "Quantity must be an integer: " + amountText;
        } catch (IllegalArgumentException error) {
            return error.getMessage();
        }
    }
    public static void main(String[] args) {
        Map<String, Integer> inventory = new HashMap<>(Map.of("diamond", 3));
        Map<String, Integer> before = new HashMap<>(inventory);
        System.out.println(process(inventory, "diamond", "-2"));
        System.out.println(inventory.equals(before));
    }
}
```

L28Example10.java · Paragraph 2/2, sequentially connected to the same file

```
        } catch (NumberFormatException error) {
            return "Quantity must be an integer: " + amountText;
        } catch (IllegalArgumentException error) {
            return error.getMessage();
        }
    }
    public static void main(String[] args) {
        Map<String, Integer> inventory = new HashMap<>(Map.of("diamond", 3));
        Map<String, Integer> before = new HashMap<>(inventory);
        System.out.println(process(inventory, "diamond", "-2"));
        System.out.println(inventory.equals(before));
    }
}
```

Output

```
The single quantity must be within 1 to 64 between: -2
true
```

Acceptance comparison table

Enter	Path	Expected
Diamond 3	Success	Quantity increased by 3
Lots of diamonds	Parsing failed	Prompt integer data unchanged
Diamond -2	Range failed	Prompt 1 to 64 data remains unchanged
Empty name 5	Verification failed	Prompt name data remains unchanged
Diamond 64	Boundary success	Allow adding

12 / 15

Challenge Blueprint Security Backpack Manager

The challenge program is called SafeInventory. Each operation returns a friendly message and the inventory is modified only if all checks are passed.

Input processing: check first, then modify

Step	Explanation
1. process	Verify text and convert results
2. addItem	Maintain inventory rules
3. Write at the end	Modify only after passing all verifications

process is responsible for turning input into operation results, and addItem is responsible for maintaining the inventory contract. Exceptions are handled where they complement the context.

Acceptance sequence

- Diamond 3 was added successfully, and the cumulative quantity is 3.
- The quantity text "many" prompt must be an integer, and the inventory remains unchanged.
- Quantity-2 prompts a single range of 1 to 64, and the inventory remains unchanged.
- Empty names are rejected before parsing and the inventory is unchanged.

- Add diamond 64 again, and the total is 67; distinguish between the single upper limit and the cumulative inventory.

13 / 15

safety backpack program

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

Specific exceptions are specifically prompted.

Element	Explanation
Single 1 to 64	Quantity range per request
Cumulative inventory	Can exceed 64, subject to the upper limit of int
Friendly results	Success or explicit rejection

Change the quantity to null, empty string, oversized integer and 65; compare the inventory before and after each operation and confirm that the status has not changed after rejecting the request.

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

```
import java.util.*;
public class SafeInventory {
    static void addItem(Map<String, Integer> inventory,
                        String item, int amount) {
        if (inventory == null)
            throw new IllegalArgumentException("Inventory cannot be null");
        if (item == null || item.isBlank())
            throw new IllegalArgumentException("Item name cannot be empty");
        if (amount < 1 || amount > 64)
            throw new IllegalArgumentException("The single quantity must be within 1 to 64 between: " +
> amount);
        String key = item.strip();
        Integer old = inventory.get(key);
        if (inventory.containsKey(key) && (old == null || old < 0))
            throw new IllegalArgumentException("The original inventory record is illegal: " + key);
        long total = (long) (old == null ? 0 : old) + amount;
        if (total > Integer.MAX_VALUE)
            throw new IllegalArgumentException("The cumulative inventory exceeds the upper limit of the
> integer: " + key);
        inventory.put(key, (int) total);
    }
    static String process(Map<String, Integer> inventory,
                        String item, String amountText) {
        if (item == null || item.isBlank()) return "Item name cannot be empty";
        if (amountText == null || amountText.isBlank()) return "Quantity cannot be empty";
        String key = item.strip();
        try {
            int amount = Integer.parseInt(amountText.strip());
            addItem(inventory, key, amount);
            return key + " Added successfully Quantity=" + inventory.get(key);
        }
    }
}
```

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

```
    } catch (NumberFormatException error) {
        return "Quantity must be an integer: " + amountText;
    } catch (IllegalArgumentException error) {
        return error.getMessage();
    }
}

public static void main(String[] args) {
    Map<String, Integer> inventory = new HashMap<>();
    System.out.println(process(inventory, "diamond", "3"));
    System.out.println(process(inventory, "diamond", "Many"));
    System.out.println(process(inventory, "diamond", "-2"));
    System.out.println(process(inventory, " ", "5"));
    System.out.println(process(inventory, "diamond", "64"));
    System.out.println("Final inventory=" + inventory.get("diamond"));
}
}
```

Output

```
diamond Added successfully Quantity=3
Quantity must be an integer: Many
The single quantity must be within 1 to 64 between: -2
Item name cannot be empty
diamond Added successfully Quantity=67
Final inventory=67
```

1 to 64 is the upper limit of the number of each operation, and the cumulative inventory can exceed 64; this is not a stacking rule for game items. Empty quantities are prompted in advance, and accumulation exceeding int will also be rejected; all writes are after checking.

14 / 15

Expansion tasks and clearance review

The job submits a successful path and at least three failed paths simultaneously. Write a sentence for each catch why it can be restored correctly.

Third-level tasks continue to strengthen error handling

Element	Explanation
Path to success	At least one true addition
Failure path	Parsing, range, empty name
catch reason	Explain why it can be restored

Lesson 29 restarts the data traversal program. We will save and load the scoreboard using Path, Files and UTF-8.

Third gear operation

- Wooden Pickaxe: Read age safely, reject non-integer and negative numbers.
- Iron Pickaxe: Analyze x y z coordinates, and retain the old position when any item is illegal.
- Diamond Pickaxe: Record the number of occurrences of each error and output a diagnostic report at the end.

Acceptance comparison table

Self-evaluation	Can	Still need to practice
I can read abnormal clues	☐	☐
I will catch specific exceptions	☐	☐
I can test it unchanged after failure	☐	☐

15 / 15

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>