

JAVA / LESSON 23 OF 40

Behavior Contracts

Abstract class skeleton, interface agreement ability

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

Make three ability contracts AbilityContracts

Bats and elytra players are not from the same family, but they can both fly. Use interfaces to stipulate "what will happen", use abstract classes to save the common skeleton, and make the capability contract small and clear. Front-end backpack: inheritance, polymorphism and method overriding. Today we do not form teams based on looks, but sign contracts based on ability.

Before you start: It is recommended to complete Lessons 21-22 first

Read the full lesson and copy the code:

<https://teacher-ma.de/minecraft-java-lesson.php?lesson=23&lang=en>

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

The common state has been written and the differential behavior is handed over to the subclasses

Element	Explanation
Actor	Abstract common skeleton
name / label	Completed part
role()	Leave it to subclasses to answer

In this lesson

01. Before starting: predict first, then let the program answer
02. Abstract classes leave spaces that must be completed
03. Abstract method forces subclass to answer
04. Is it possible to only describe the interface?
05. The implementation method must be disclosed
06. A class can sign multiple contracts
07. Interface reference calling ability
08. The small interface is clearer than the universal sundry bag
09. Comparison of abstract classes and interfaces
10. Contract Testing and Failure Clinic
11. Challenge Blueprint Three Abilities
12. Challenges Implementing Capability Contracts
13. Expansion tasks and clearance review
14. Own acceptance record

01 / 14

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 / 14

Abstract classes leave spaces that must be completed

If all roles have names but there is no unified role implementation, the parent class should not pretend to give a meaningless answer. Abstract classes allow common parts to be written and methods to be explicitly left unfinished.

The common state has been written and the differential behavior is handed over to the subclasses

Element	Explanation
Actor	Abstract common skeleton
name / label	Completed part
role()	Leave it to subclasses to answer

`new Actor()` will fail to compile because the abstract method has not been implemented yet. A concrete subclass must be created to complete the contract.

Fault experiment: The abstract class `Actor` cannot be new directly; it also leaves the implementation task of `role`. Errors are intentionally left below, please anticipate them first and then try to fix them.

BadAbstract.java · Intentional error example

```
abstract class Actor {
    protected final String name;
    Actor(String name) { this.name = name; }
    abstract String role();
    public String label() { return role() + " " + name; }
}

public class BadAbstract {
    public static void main(String[] args) { Actor a = new Actor("Not completed"); }
```

03 / 14

Abstract method forces subclass to answer

Abstract methods are like required questions on a test paper. If the subclass is still a concrete class, it must give a method body with a correct signature.

There is no method body but the name parameters and return type are complete

Element	Explanation
Concrete subclass	Abstract methods must be completed
@Override	Verify method signature
Bat	Provide bat character

Return type, method name, and parameters must match. Leave @Override on and let the compiler check the answer sheet for you.

L23Example02.java · Complete independent example

```
abstract class Actor {
    protected final String name;
    Actor(String name) { this.name = name; }
    abstract String role();
    public String label() { return role() + " " + name; }
}
class Bat extends Actor {
    Bat(String name) { super(name); }
    @Override String role() { return "bat"; }
}
public class L23Example02 {
    public static void main(String[] args) {
        Actor a = new Bat("Night shadow");
        System.out.println(a.label());
    }
}
```

Output

```
bat Night shadow
```

Failure experiment: non-abstract Bat does not complete role() and the compiler rejects it. Errors are intentionally left below, please anticipate them first and then try to fix them.

BadRole.java · Intentional error example

```
abstract class Actor {
    protected final String name;
    Actor(String name) { this.name = name; }
    abstract String role();
    public String label() { return role() + " " + name; }
}
class Bat extends Actor { Bat(String name) { super(name); } }
public class BadRole { public static void main(String[] args) { } }
```

04 / 14

Is it possible to only describe the interface?

The interface is suitable for agreeing on cross-family capabilities. Bats are animals and elytra players are players, but both can implement Flyable; the caller only needs to know that the other can fly.

The same family is not required, only the same ability is required

Element	Explanation
Flyable	Agreed flight capability
Bat	Realize flight
ElytraPlayer	Also realize flight

extends answers "what", implements answers "what". Don't mix the two issues in one inheritance chain.

L23Example03.java · Complete independent example

```
interface Flyable { String fly(); }
class Bat implements Flyable {
    @Override public String fly() { return "Bat flight"; }
}
class ElytraPlayer implements Flyable {
    @Override public String fly() { return "Player gliding"; }
}
public class L23Example03 {
    public static void main(String[] args) {
        Flyable[] team = {new Bat(), new ElytraPlayer()};
        for (Flyable member : team) System.out.println(member.fly());
    }
}
```

Output

```
Bat flight
Player gliding
```

This chapter introduces the abstract instance methods of interfaces. Interfaces can also declare default, static, and private methods; "only describing the contract" is a design purpose, not a Java prohibition on method bodies.

05 / 14

The implementation method must be disclosed

Interface methods are public by default. Implementing classes must provide the same visible access using public.

The interface commitment is a public entrance

Element	Explanation
Interface abstract method	Default public
Implementation method	Cannot reduce visibility
@Override	Check the same contract

The abstract instance methods of the interface in this section are public by default; the implementation methods must also be public. Interfaces can contain default, static and private methods, and these methods cannot all be described as abstract methods.

Fault experiment: The interface fly() is public; the implementation method cannot reduce the visibility. Errors are intentionally left below, please anticipate them first and then try to fix them.

BadPublic.java · Intentional error example

```
interface Flyable { String fly(); }  
class Bat implements Flyable { String fly() { return "Flight"; } }  
public class BadPublic { public static void main(String[] args) { } }
```

Add public before String fly() and add @Override. Interface fields are constants and cannot replace each object's own mutable state.

06 / 14

A class can sign multiple contracts

A Java class can only directly inherit one parent class, but can implement multiple interfaces. A flying bomb can be both Flyable and Explodable.

There is only one parent class for inheritance, and there can be multiple interfaces.

Element	Explanation
A parent class	A class can only directly inherit one
Multiple interfaces	Capabilities can be combined
FlyingMine	Can fly and detonate at the same time

The interface is like a composable capability card. Classes only receive contracts that they can actually complete, and there is no need to create complex family trees for multiple inheritance.

L23Example05.java · Complete independent example

```
interface Flyable { String fly(); }
interface Explodable { String explode(); }
class FlyingMine implements Flyable, Explodable {
    @Override public String fly() { return "Fly to target"; }
    @Override public String explode() { return "Detonate"; }
}
public class L23Example05 {
    public static void main(String[] args) {
        FlyingMine mine = new FlyingMine();
        System.out.println(mine.fly());
        System.out.println(mine.explode());
    }
}
```

Output

```
Fly to target
Detonate
```

07 / 14

Interface reference calling ability

If the method parameters are written in the interface type, any implementer can be accepted. This is the expansion of polymorphism from family relationships to capability relationships.

Tool methods only rely on required contracts

Element	Explanation
Interface parameters	As long as the required capabilities
Any implementer	Acceptable same trigger
null	This lesson agrees to skip safely

Only the fly method is needed to receive Flyable, and there is no need to receive a larger Actor or specific Bat.

L23Example06.java · Section 1/2, sequentially connected to the same file

```
interface Flyable { String fly(); }
interface Tradable { String trade(String item); }
interface Explodable { int blastPower(); String explode(); }
abstract class Actor {
    protected final String name;
    Actor(String name) { this.name = name; }
    abstract String role();
    public String label() { return role() + " " + name; }
}
class Bat extends Actor implements Flyable {
    Bat(String name) { super(name); }
    @Override String role() { return "bat"; }
    @Override public String fly() { return name + " Fly to high places"; }
}
class Villager extends Actor implements Tradable {
    Villager(String name) { super(name); }
    @Override String role() { return "Villagers"; }
    @Override public String trade(String item) {
        return item == null || item.isBlank()
            ? name + " Reject empty transaction" : name + " Accept" + item.strip();
    }
}
class Creeper extends Actor implements Explodable {
    Creeper(String name) { super(name); }
    @Override String role() { return "Creeper"; }
    @Override public int blastPower() { return 3; }
    @Override public String explode() { return name + " Explosion"; }
}
```

L23Example06.java · Paragraph 2/2, sequentially connected to the same file

```
public class L23Example06 {
    static void triggerFlight(Flyable target) {
        if (target != null) System.out.println(target.fly());
    }
    static void triggerTrade(Tradable target, String item) {
        if (target != null) System.out.println(target.trade(item));
    }
    static void triggerExplosion(Explodable target) {
        if (target != null)
            System.out.println(target.explode() + " Strength=" + target.blastPower());
    }
    public static void main(String[] args) {
        triggerFlight(new Bat("Night shadow"));
        triggerTrade(new Villager("Village chief"), "Emerald");
        triggerExplosion(new Creeper("Mung beans"));
        triggerFlight(null);
    }
}
```

Output

```
Night shadow Fly to high places
Village chief AcceptEmerald
Mung beans Explosion Strength=3
```

08 / 14

The small interface is clearer than the universal sundry bag

Packing flying, trading, and explosions into one MegaAbility forces villagers to fly that they can't. Small interfaces are easier to understand and compose.

Each contract revolves around a stable capability

Element	Explanation
Flyable	Only responsible for flying
Tradable	Only responsible for transactions
Explodable	Only responsible for explosion

Ask whether each method is always used by the same batch of classes. If the answer is no, break the contract into smaller capability interfaces.

L23Example07.java · Complete independent example

```
interface Flyable { String fly(); }
interface Tradable { String trade(String item); }
interface Explodable { int blastPower(); String explode(); }
abstract class Actor {
    protected final String name;
    Actor(String name) { this.name = name; }
    abstract String role();
    public String label() { return role() + " " + name; }
}
class Bat extends Actor implements Flyable {
    Bat(String name) { super(name); }
    @Override String role() { return "bat"; }
    @Override public String fly() { return name + " Fly to high places"; }
}
class Villager extends Actor implements Tradable {
    Villager(String name) { super(name); }
    @Override String role() { return "Villagers"; }
    @Override public String trade(String item) {
        return item == null || item.isBlank()
            ? name + " Reject empty transaction" : name + " Accept" + item.strip();
    }
}
class Creeper extends Actor implements Explodable {
    Creeper(String name) { super(name); }
    @Override String role() { return "Creeper"; }
    @Override public int blastPower() { return 3; }
    @Override public String explode() { return name + " Explosion"; }
}
public class L23Example07 {
    public static void main(String[] args) {
        Tradable merchant = new Villager("Village chief");
        System.out.println(merchant.trade(""));
    }
}
```

Output

```
Village chief Reject empty transaction
```

Villagers only sign Tradable; no need to pretend to fly. Methods with the same name do not automatically become interface implementations, they must implements and satisfy the signature.

09 / 14

Comparison of abstract classes and interfaces

Abstract classes can save instance state, constructors and common implementations; interfaces emphasize the ability that callers can rely on. A class often inherits an abstract parent class and implements several interfaces at the same time.

Family skeleton and capability contracts solve different problems

Element	Explanation
Abstract class	State, constructor, common implementation
Interface	Caller-dependent capabilities
Use in combination	Inherit the skeleton and implement the interface

Bat extends Actor implements Flyable simultaneously answers what it is and what it does.

Acceptance comparison table

Problem	Abstract class	Interface
Expression	Same family skeleton	Cross-family capabilities
Instance field	Yes	Do not save normal instance state
Constructor	Yes	No
Quantity	Only one can be inherited	Can realize multiple
Typical example	Actor	Flyable

10 / 14

Contract Testing and Failure Clinic

Interface errors are often exposed during the compilation phase, and behavioral errors still need to be discovered through testing.

Compiler checks signature Run test checks behavior

Element	Explanation
Signature matching	Parameters cannot be omitted
Empty transaction	Explicitly return rejection
Explosion intensity	This lesson agrees 3

First check whether public is missing, then check the method signature and `@Override`, and finally run the behavioral test of each contract.

Fault experiment: `trade()` lacks the `String` parameter and cannot complete the `trade(String)` contract. Errors are intentionally left below, please anticipate them first and then try to fix them.

BadTrade.java · Intentional error example

```
interface Tradable { String trade(String item); }
class Villager implements Tradable {
    public String trade() { return "Transaction"; }
}
public class BadTrade { public static void main(String[] args) { } }
```

Acceptance comparison table

Test	Enter	Expected
Flight	Bat	Return to flight description
Transaction	Empty item name	Return rejection prompt
Explosion	Creeper	Strength is 3
Null reference	null	Safety prompt or skip

11 / 14

Challenge Blueprint Three Abilities

The challenge program is called AbilityContracts. Bat can fly, Villager can trade, and Creeper can explode; all three inherit Actor and realize their respective abilities.

Abstract roles provide names and three interfaces provide capability entrances.

Element	Explanation
Bat	Flight Contract
Villager	Transaction contract
Creeper	Explosion Contract

First declare three interfaces and Actors, then implement specific classes one by one, and finally write the trigger method. Compile it every time you sign a contract.

Acceptance conditions

- Actor cannot be new directly.
- All three concrete classes implement role.
- All interface implementation methods are public.
- The trigger method uses interface parameters.
- A rejection prompt is returned when trading an empty string.

12 / 14

Challenges Implementing Capability Contracts

Create AbilityContracts.java, compile and run using Java 21.

Flight transaction explosion is triggered uniformly through the interface

Element	Explanation
First statement	Three interfaces and Actor
Reimplementation	Complete specific classes one by one
Post trigger	Interface parameters are uniformly called

Let Bat implement the second interface again; remove public and observe compilation errors; confirm that the trigger method does not depend on the specific class.

AbilityContracts.java · Paragraph 1/2, sequentially connected to the same file

```
interface Flyable { String fly(); }
interface Tradable { String trade(String item); }
interface Explodable { int blastPower(); String explode(); }
abstract class Actor {
    protected final String name;
    Actor(String name) { this.name = name; }
    abstract String role();
    public String label() { return role() + " " + name; }
}
class Bat extends Actor implements Flyable {
    Bat(String name) { super(name); }
    @Override String role() { return "bat"; }
    @Override public String fly() { return name + " Fly to high places"; }
}
class Villager extends Actor implements Tradable {
    Villager(String name) { super(name); }
    @Override String role() { return "Villagers"; }
    @Override public String trade(String item) {
        return item == null || item.isBlank()
            ? name + " Reject empty transaction" : name + " Accept" + item.strip();
    }
}
class Creeper extends Actor implements Explodable {
    Creeper(String name) { super(name); }
    @Override String role() { return "Creeper"; }
    @Override public int blastPower() { return 3; }
    @Override public String explode() { return name + " Explosion"; }
}
```

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

```
public class AbilityContracts {
    static void triggerFlight(Flyable target) {
        if (target != null) System.out.println(target.fly());
    }
    static void triggerTrade(Tradable target, String item) {
        if (target != null) System.out.println(target.trade(item));
    }
    static void triggerExplosion(Explodable target) {
        if (target != null)
            System.out.println(target.explode() + " Strength=" + target.blastPower());
    }

    public static void main(String[] args) {
        Bat bat = new Bat("Night shadow");
        Villager villager = new Villager("Village chief");
        Creeper creeper = new Creeper("Mung beans");
        System.out.println(bat.label());
        triggerFlight(bat);
        triggerTrade(villager, "Emerald");
        triggerTrade(villager, " ");
        triggerExplosion(creeper);
        triggerFlight(null);
    }
}
```

Output

```
bat Night shadow
Night shadow Fly to high places
Village chief AcceptEmerald
Village chief Reject empty transaction
Mung beans Explosion Strength=3
```

13 / 14

Expansion tasks and clearance review

Each interface must be accompanied by a contract description, clearly stating the input, return value and failure handling.

Three-tier mission expansion capability contract

Element	Explanation
Contract description	Write clear input and output
Failure Rules	Explicitly reject or skip
Compilation and Behavior	Both inspections are required

Lesson 24: Limit the direction to enum and load the coordinates into record. Moving does not secretly change the old coordinates, but returns a new position.

Third gear operation

- Wooden Pickaxe: Designed to be Swimmable so that both dolphins and players can swim.
- Iron Pick: Let a class implement Flyable and Tradable at the same time, and use two interface references to call.
- Diamond Pickaxe: Design the redstone device interface, implement doors, lights and traps respectively, and write unified triggers.

Acceptance comparison table

Self-evaluation	Can	Still need to practice
I can differentiate between abstract classes and interfaces	☐	☐
I will use implements	☐	☐
I can design small interfaces	☐	☐

14 / 14

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>