Chapter 4.5: Abstract Classes and Interfaces

Object Oriented Programming with Java

Abstraction | Abstract classes | Abstract methods | Interfaces

implements and multiple interfaces | default, static, private methods

Lambdas and functional interfaces | sealed, non-sealed, permits

Interface or abstract class? | Design patterns

Sercan KÜLCÜ · Giresun University · Computer Engineering

© Sercan Külcü. All rights reserved. · Interactive HTML version of these slides:
https://sercankulcu.github.io/files/java/slides/Chapter_04_05_Abstract_Classes_and_Interfaces.html

Learning Outcomes

Abstract class

Declare `abstract` classes and methods; explain why an abstract class cannot be instantiated and what a subclass must implement.

Template method

Trace code to show a concrete method of an abstract class calling an abstract method implemented in a subclass.

Interface

Declare an `interface`, implement one or more interfaces with `implements`, and use `default`, `static` and `private` methods.

Lambda

Recognize a functional interface; write single-method behavior as a lambda expression and trace when the lambda runs.

Design patterns

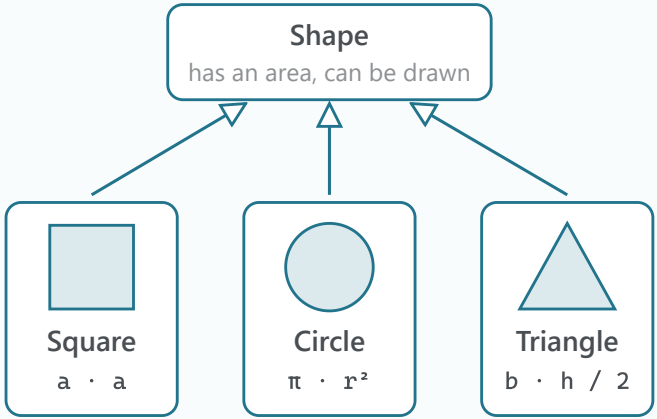
Explain which problem Singleton, Factory, Observer, Strategy and MVC solve and how they rely on interfaces.

Design decision

Recognize `sealed` types; decide whether a problem calls for an interface, an abstract class, or both; use generic types and annotations.

Abstraction

- **Abstraction**: hiding complex details and showing only the **essential features**.
- A driver knows the steering wheel and the pedals; they need not know what goes on inside the engine.
- “Shape” is a concept: every shape has an area, but the area of a **generic shape** cannot be computed. Only Square, Circle and Triangle know their formulas.
- Java has two tools for abstraction: **abstract classes** (`abstract class`) and **interfaces** (`interface`).
- Both say **what** to do and leave **how** to the subclasses.



The common concept on top, the details below.

Abstract Class

- Declared with the `abstract` keyword: `public abstract class Shape`.
- May contain both **abstract** (bodiless) and **concrete** (with a body) methods.

```
public abstract class Shape {
    // concrete (non-abstract) method
    void move(int x, int y) {
        System.out.println("Moved to (" + x + ", " + y + ")");
    }
    // abstract method (subclasses must implement)
    abstract void draw();
}
```

Concrete method: `move`

Its body is ready. Subclasses **inherit** it as is, or override it if they want.

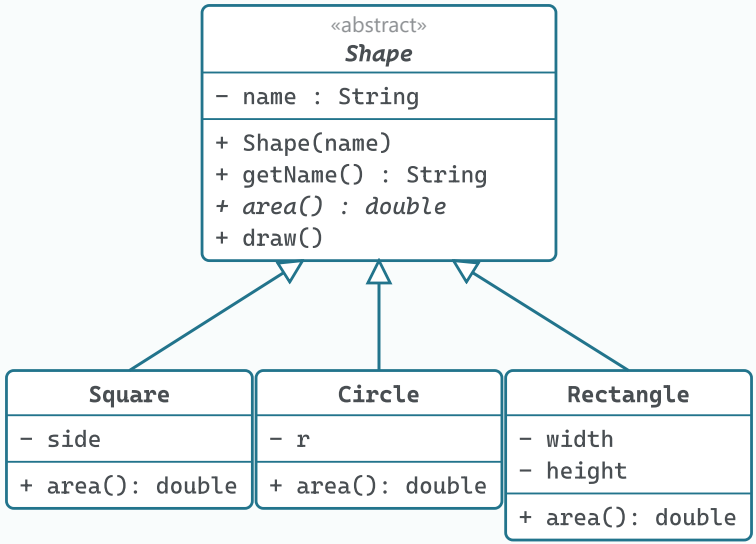
Abstract method: `draw`

It has only a signature. Every concrete subclass **must write** its own `draw()` body.

An abstract class is like an unfinished floor plan: some rooms are done, others only have their place marked.

Properties of an Abstract Class

- An abstract class **cannot be instantiated** (it has no instances).
- Other classes extend it with `extends` (inheritance).
- A **concrete** subclass must **implement** every abstract method.
- It gathers shared state (fields) and behavior in one place → code is **reused**.
- Subclasses fill in the abstract methods to suit their needs.
- It can have **constructors**; a subclass calls them with `super(...)`.



In UML, abstract classes and abstract methods are written in *italics*.

An Abstract Class Cannot Be Instantiated

```
Shape s1 = new Shape("?");      // ERROR
Shape s2 = new Square(3);         // OK: abstract variable type
Shape[] arr = new Shape[3];    // OK: the array holds only references
```

Compiler message (line 1)

```
error: Shape is abstract; cannot be instantiated
```

Why forbidden?

`Shape`'s `area()` has no body. If `new Shape()` were allowed, the call `s1.area()` would have no code to run.

Fine as a reference type

Variables, parameters, array elements and return types can be `Shape`. `new Shape[3]` creates 3 empty (`null`) references, not `Shape` objects.

You cannot write `abstract final class`: `final` forbids extension, `abstract` requires it → `illegal combination of modifiers: abstract and final`.

Abstract Methods

- Declared **without an implementation**.
- No braces; ends with a **semicolon**:

```
public abstract void draw();
```

- A class with an abstract method **must itself be abstract**.
- Abstract classes usually sit at the **top** of an inheritance hierarchy.

```
public abstract class Shape {
    // abstract method
    public abstract void draw();
}

// in a separate file:
public class Circle extends Shape {
    // must implement draw
    public void draw() {
        System.out.println("Drawing a circle");
    }
}
```

Output of `new Circle().draw();`

```
Drawing a circle
```

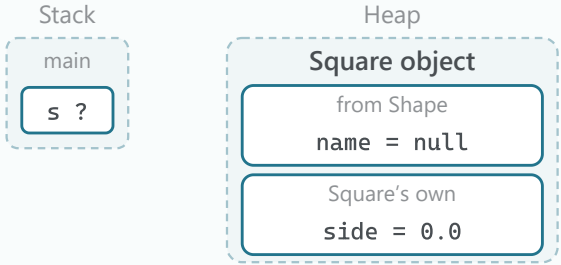
Rules for Abstract Methods

Code	Result
`abstract void draw();` (in an abstract class)	✓ OK
`abstract void draw() { }`	`abstract methods cannot have a body`
`class Shape { abstract void draw(); }`	`Shape is not abstract and does not override abstract method draw() in Shape`
`class Circle extends Shape { }` (no draw)	`Circle is not abstract and does not override abstract method draw() in Shape`
`private abstract`, `static abstract`, `final abstract`	`illegal combination of modifiers: abstract and private` (same for static, final)
`abstract class Circle extends Shape { }`	✓ OK: an abstract subclass may leave the implementation to its own subclasses
`abstract class A { void f() { } }`	✓ OK: an abstract class need not have any abstract methods (still, `new A()` is not allowed)

Why not `private` / `static` / `final`? All three prevent overriding in a subclass, and an abstract method exists to be overridden.

`new Square(3)`: The Constructor Chain

```
1  abstract class Shape {
2      private String name;
3      Shape(String name) {
4          this.name = name;
5          System.out.println(name + " created");
6      }
7      abstract double area();
8  }
9  class Square extends Shape {
10     private double side;
11     Square(double side) {
12         super("Square");
13         this.side = side;
14     }
15     double area() { return side * side; }
16 }
17 // in main:
▶  Shape s = new Square(3);
19 System.out.println(s.area());
```



`new Square(3)`: first a **Square** object is allocated on the heap. Fields get default values: `null` and `0.0`.

Step 1 / 10

CALL STACK

`main()`

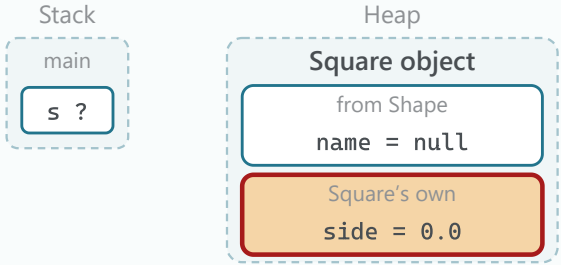
VARIABLES

`s = –`

OUTPUT

`new Square(3)`: The Constructor Chain

```
1  abstract class Shape {
2      private String name;
3      Shape(String name) {
4          this.name = name;
5          System.out.println(name + " created");
6      }
7      abstract double area();
8  }
9  class Square extends Shape {
10     private double side;
▶      Square(double side) {
12         super("Square");
13         this.side = side;
14     }
15     double area() { return side * side; }
16 }
17 // in main:
18 Shape s = new Square(3);
19 System.out.println(s.area());
```



Square's constructor runs. The parameter is `side = 3.0` (the `int` 3 widened to `double`).

Step 2 / 10

CALL STACK

- `Square(3)`
- `main()`

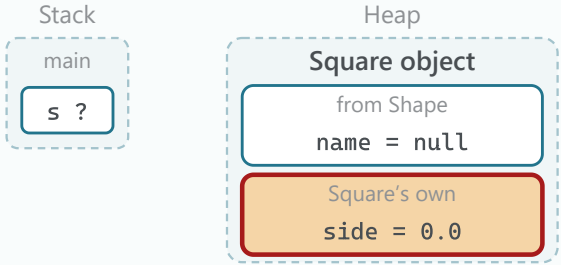
VARIABLES

`side = 3.0`

OUTPUT

`new Square(3)`: The Constructor Chain

```
1  abstract class Shape {
2      private String name;
3      Shape(String name) {
4          this.name = name;
5          System.out.println(name + " created");
6      }
7      abstract double area();
8  }
9  class Square extends Shape {
10     private double side;
11     Square(double side) {
▶          super("Square");
13         this.side = side;
14     }
15     double area() { return side * side; }
16 }
17 // in main:
18 Shape s = new Square(3);
19 System.out.println(s.area());
```



The first step is `super("Square")` : the superclass constructor is called. An abstract class can have a constructor too.

Step 3 / 10

CALL STACK

- `Square(3)`
- `main()`

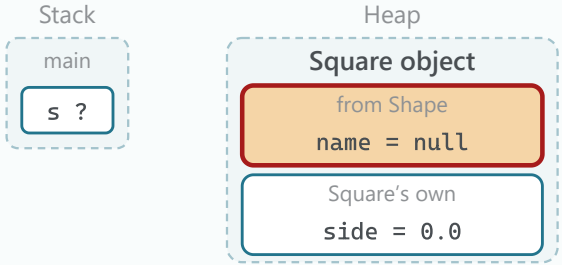
VARIABLES

`side = 3.0`

OUTPUT

`new Square(3)`: The Constructor Chain

```
1  abstract class Shape {
2      private String name;
3      Shape(String name) {
4          this.name = name;
5          System.out.println(name + " created");
6      }
7      abstract double area();
8  }
9  class Square extends Shape {
10     private double side;
11     Square(double side) {
12         super("Square");
13         this.side = side;
14     }
15     double area() { return side * side; }
16 }
17 // in main:
18 Shape s = new Square(3);
19 System.out.println(s.area());
```



The Shape constructor runs. **No** separate Shape object is created: the Shape part of the same object is being built.

Step 4 / 10

CALL STACK

- `Shape("Square")`
- `Square(3)`
- `main()`

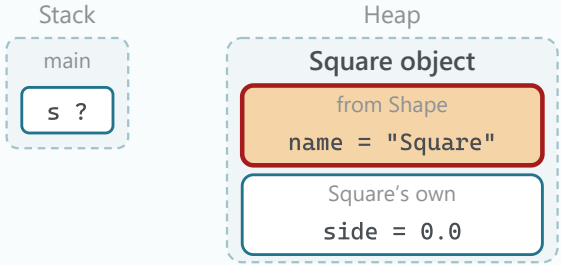
VARIABLES

`name = "Square"`

OUTPUT

`new Square(3)`: The Constructor Chain

```
1  abstract class Shape {
2      private String name;
3      Shape(String name) {
▶          this.name = name;
5          System.out.println(name + " created");
6      }
7      abstract double area();
8  }
9  class Square extends Shape {
10     private double side;
11     Square(double side) {
12         super("Square");
13         this.side = side;
14     }
15     double area() { return side * side; }
16 }
17 // in main:
18 Shape s = new Square(3);
19 System.out.println(s.area());
```



`this.name = name` : the object's `name` field becomes "Square".

Step 5 / 10

CALL STACK

- **`Shape("Square")`**
- `Square(3)`
- `main()`

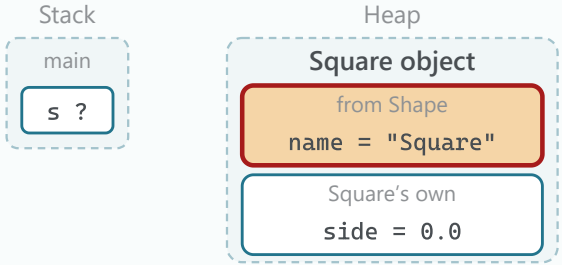
VARIABLES

`name` = **`"Square"`**

OUTPUT

`new Square(3)`: The Constructor Chain

```
1  abstract class Shape {
2      private String name;
3      Shape(String name) {
4          this.name = name;
▶          System.out.println(name + " created");
6      }
7      abstract double area();
8  }
9  class Square extends Shape {
10     private double side;
11     Square(double side) {
12         super("Square");
13         this.side = side;
14     }
15     double area() { return side * side; }
16 }
17 // in main:
18 Shape s = new Square(3);
19 System.out.println(s.area());
```



The message is printed. The Shape constructor ends and its frame leaves the stack.

Step 6 / 10

CALL STACK

- `Shape("Square")`
- `Square(3)`
- `main()`

VARIABLES

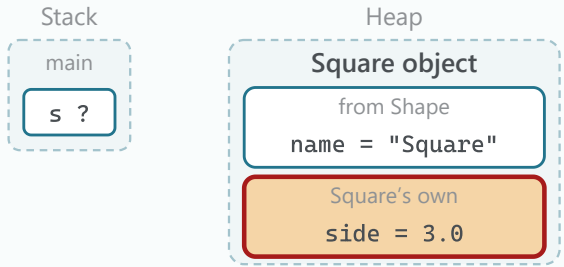
`name = "Square"`

OUTPUT

```
Square created
```

`new Square(3)`: The Constructor Chain

```
1  abstract class Shape {
2      private String name;
3      Shape(String name) {
4          this.name = name;
5          System.out.println(name + " created");
6      }
7      abstract double area();
8  }
9  class Square extends Shape {
10     private double side;
11     Square(double side) {
12         super("Square");
▶           this.side = side;
14     }
15     double area() { return side * side; }
16 }
17 // in main:
18 Shape s = new Square(3);
19 System.out.println(s.area());
```



Back in Square's constructor: `this.side = 3.0`. The object is now fully built.

Step 7 / 10

CALL STACK

- **Square(3)**
- main()

VARIABLES

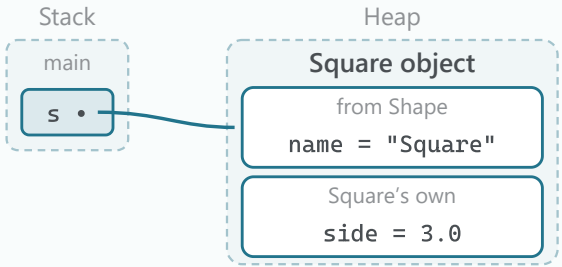
side = **3.0**

OUTPUT

```
Square created
```

`new Square(3)`: The Constructor Chain

```
1  abstract class Shape {
2      private String name;
3      Shape(String name) {
4          this.name = name;
5          System.out.println(name + " created");
6      }
7      abstract double area();
8  }
9  class Square extends Shape {
10     private double side;
11     Square(double side) {
12         super("Square");
13         this.side = side;
14     }
15     double area() { return side * side; }
16 }
17 // in main:
▶  Shape s = new Square(3);
19 System.out.println(s.area());
```



The object's reference is stored in `s`. The variable's type is **Shape** (abstract); the object it points to is a **Square**.

Step 8 / 10

CALL STACK

`main()`

VARIABLES

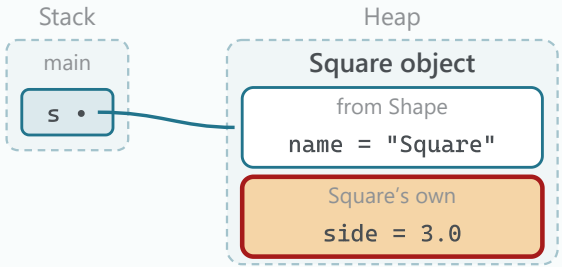
`s = → Square`

OUTPUT

```
Square created
```

`new Square(3)`: The Constructor Chain

```
1  abstract class Shape {
2      private String name;
3      Shape(String name) {
4          this.name = name;
5          System.out.println(name + " created");
6      }
7      abstract double area();
8  }
9  class Square extends Shape {
10     private double side;
11     Square(double side) {
12         super("Square");
13         this.side = side;
14     }
▶      double area() { return side * side; }
16 }
17 // in main:
18 Shape s = new Square(3);
19 System.out.println(s.area());
```



`s.area()` : `area()` is abstract in Shape. At run time the object's type decides → **Square.area()**: 3.0 · 3.0.

Step 9 / 10

CALL STACK

- `Square.area()`
- `main()`

VARIABLES

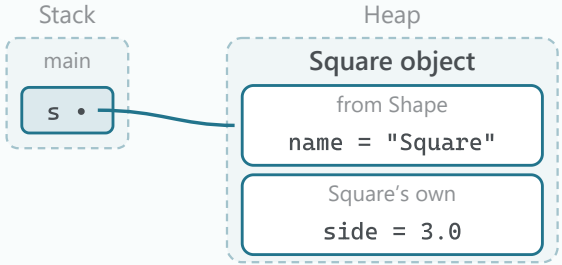
`side = 3.0`

OUTPUT

```
Square created
```

`new Square(3)`: The Constructor Chain

```
1  abstract class Shape {
2      private String name;
3      Shape(String name) {
4          this.name = name;
5          System.out.println(name + " created");
6      }
7      abstract double area();
8  }
9  class Square extends Shape {
10     private double side;
11     Square(double side) {
12         super("Square");
13         this.side = side;
14     }
15     double area() { return side * side; }
16 }
17 // in main:
18 Shape s = new Square(3);
▶  System.out.println(s.area());
```



The returned `9.0` is printed.

Step 10 / 10

CALL STACK

`main()`

VARIABLES

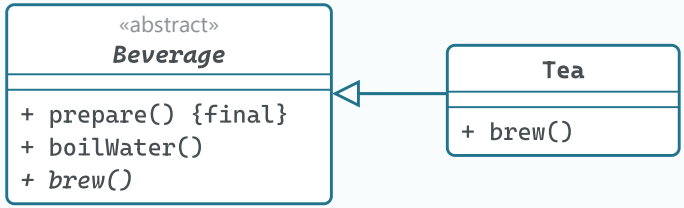
`s = → Square`

OUTPUT

```
Square created
9.0
```

A Concrete Method Calls an Abstract Method

```
1  abstract class Beverage {
2      final void prepare() {
3          boilWater();
4          brew();
5          System.out.println("Served");
6      }
7      void boilWater() {
8          System.out.println("Water boiled");
9      }
10     abstract void brew();
11 }
12 class Tea extends Beverage {
13     void brew() {
14         System.out.println("Tea brewed");
15     }
16 }
17 // in main:
▶  Beverage i = new Tea();
19 i.prepare();
```



The variable type is `Beverage` (abstract), the object a **Tea**. Tea is concrete because it implements the only abstract method, `brew()`.

Step 1 / 8

CALL STACK

`main()`

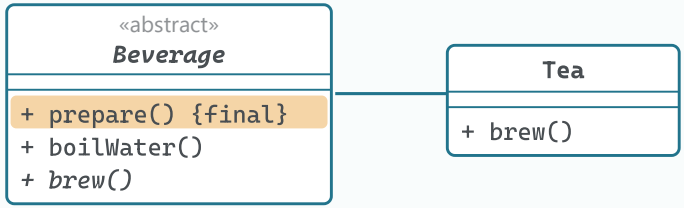
VARIABLES

`i = → Tea`

OUTPUT

A Concrete Method Calls an Abstract Method

```
1  abstract class Beverage {
▶      final void prepare() {
3          boilWater();
4          brew();
5          System.out.println("Served");
6      }
7      void boilWater() {
8          System.out.println("Water boiled");
9      }
10     abstract void brew();
11 }
12 class Tea extends Beverage {
13     void brew() {
14         System.out.println("Tea brewed");
15     }
16 }
17 // in main:
18 Beverage i = new Tea();
19 i.prepare();
```



Tea has no `prepare()` → the concrete method inherited from Beverage runs. `final`: subclasses cannot change this order.

Step 2 / 8

CALL STACK

- `prepare()`
- `main()`

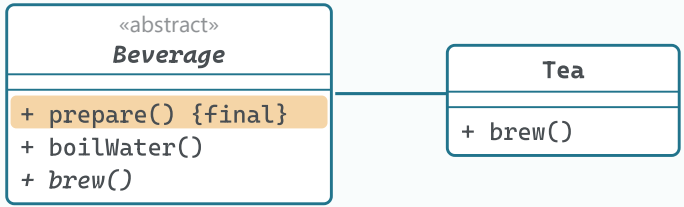
VARIABLES

`this = → Tea`

OUTPUT

A Concrete Method Calls an Abstract Method

```
1  abstract class Beverage {
2      final void prepare() {
▶          boilWater();
4          brew();
5          System.out.println("Served");
6      }
7      void boilWater() {
8          System.out.println("Water boiled");
9      }
10     abstract void brew();
11 }
12 class Tea extends Beverage {
13     void brew() {
14         System.out.println("Tea brewed");
15     }
16 }
17 // in main:
18 Beverage i = new Tea();
19 i.prepare();
```



Step 1: `boilWater()`, that is `this.boilWater()`, is called.

Step 3 / 8

CALL STACK

- **prepare()**
- main()

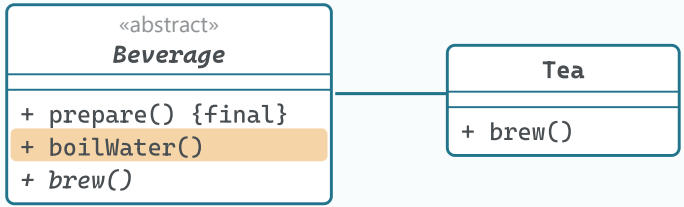
VARIABLES

this = → **Tea**

OUTPUT

A Concrete Method Calls an Abstract Method

```
1  abstract class Beverage {
2      final void prepare() {
3          boilWater();
4          brew();
5          System.out.println("Served");
6      }
7      void boilWater() {
▶          System.out.println("Water boiled");
9      }
10     abstract void brew();
11 }
12 class Tea extends Beverage {
13     void brew() {
14         System.out.println("Tea brewed");
15     }
16 }
17 // in main:
18 Beverage i = new Tea();
19 i.prepare();
```



The concrete `boilWater()` in Beverage runs.

Step 4 / 8

CALL STACK

- **boilWater()**
- prepare()
- main()

VARIABLES

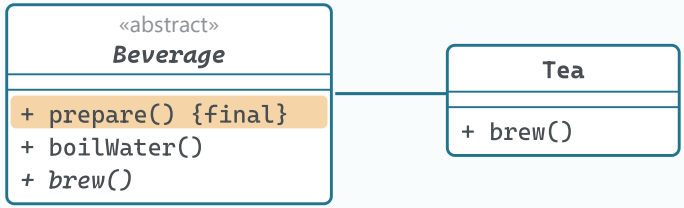
this = → **Tea**

OUTPUT

```
Water boiled
```

A Concrete Method Calls an Abstract Method

```
1  abstract class Beverage {
2      final void prepare() {
3          boilWater();
▶          brew();
5          System.out.println("Served");
6      }
7      void boilWater() {
8          System.out.println("Water boiled");
9      }
10     abstract void brew();
11 }
12 class Tea extends Beverage {
13     void brew() {
14         System.out.println("Tea brewed");
15     }
16 }
17 // in main:
18 Beverage i = new Tea();
19 i.prepare();
```



Step 2: `brew()`. It has no body in Beverage, but `this` is a Tea object...

Step 5 / 8

CALL STACK

- **prepare()**
- main()

VARIABLES

this = → **Tea**

OUTPUT

```
Water boiled
```

A Concrete Method Calls an Abstract Method

```
1  abstract class Beverage {
2      final void prepare() {
3          boilWater();
4          brew();
5          System.out.println("Served");
6      }
7      void boilWater() {
8          System.out.println("Water boiled");
9      }
10     abstract void brew();
11 }
12 class Tea extends Beverage {
13     void brew() {
▶          System.out.println("Tea brewed");
15     }
16 }
17 // in main:
18 Beverage i = new Tea();
19 i.prepare();
```



...so **Tea.brew()** runs. Code in the abstract class can call subclass code that has not even been written yet.

Step 6 / 8

CALL STACK

- Tea.brew()
- prepare()
- main()

VARIABLES

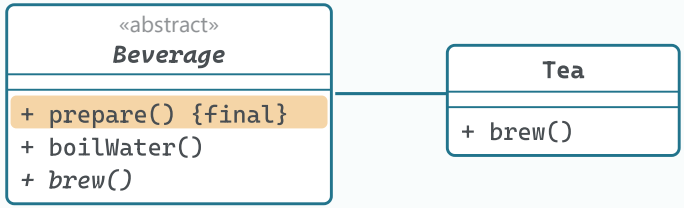
this = → Tea

OUTPUT

```
Water boiled
Tea brewed
```

A Concrete Method Calls an Abstract Method

```
1  abstract class Beverage {
2      final void prepare() {
3          boilWater();
4          brew();
▶          System.out.println("Served");
6      }
7      void boilWater() {
8          System.out.println("Water boiled");
9      }
10     abstract void brew();
11 }
12 class Tea extends Beverage {
13     void brew() {
14         System.out.println("Tea brewed");
15     }
16 }
17 // in main:
18 Beverage i = new Tea();
19 i.prepare();
```



Step 3: the serving message is printed.

Step 7 / 8

CALL STACK

- `prepare()`
- `main()`

VARIABLES

`this = → Tea`

OUTPUT

```
Water boiled
Tea brewed
Served
```

A Concrete Method Calls an Abstract Method

```
1  abstract class Beverage {
2      final void prepare() {
3          boilWater();
4          brew();
5          System.out.println("Served");
6      }
7      void boilWater() {
8          System.out.println("Water boiled");
9      }
10     abstract void brew();
11 }
12 class Tea extends Beverage {
13     void brew() {
14         System.out.println("Tea brewed");
15     }
16 }
17 // in main:
18 Beverage i = new Tea();
▶  i.prepare();
```



The skeleton lives in the abstract class, the varying step in the subclass: the **template method** pattern. A `Coffee` class writes only `brew()`.

Step 8 / 8

CALL STACK

`main()`

VARIABLES

`i = → Tea`

OUTPUT

```
Water boiled
Tea brewed
Served
```

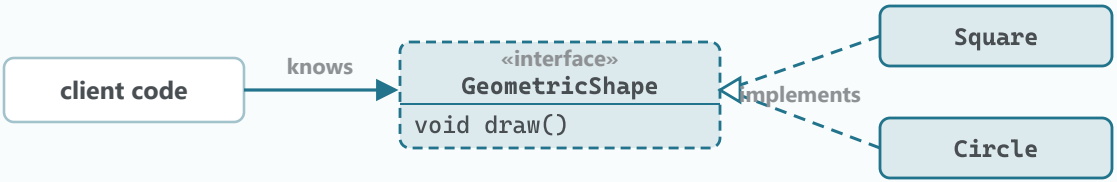
What Is an Interface?

- A **contract** guaranteeing that a system provides certain operations.
- It is the **integration point** between two systems: the client knows only the interface.
- Declared with the `interface` keyword.
- Interface methods are **public**, whether or not you write `public`.
- Unless it is `static`, `default` or `private`, a method is implicitly **abstract**.
- The only fields allowed are `public static final` **constants**.

Analogy: socket and plug

A wall socket is a contract: whatever you plug in gets the same voltage. An iron and a computer both follow the socket standard; the grid knows nothing about the device's insides, and vice versa.

An abstract class sets up an "**is a kind of**" (is-a) relationship; an interface defines a "**can do**" capability.



Declaring an Interface

What we write

```
public interface GeometricShape {
    // abstract (public abstract)
    void draw();
    // constant (public static final)
    int SIDES = 4;
}
```

What the compiler sees

```
public interface GeometricShape {
    public abstract void draw();
    public static final int SIDES = 4;
}
```

- Constants are accessed through the interface name: `GeometricShape.SIDES` → `4`. Their names are written in **UPPER_CASE**.
- A constant must be initialized: `int value;` → `error: = expected`.
- A constant cannot be changed: `GeometricShape.SIDES = 5;` → `error: cannot assign a value to static final variable SIDES`.
- An interface cannot be instantiated either: `new GeometricShape()` → `error: GeometricShape is abstract; cannot be instantiated`.

Implementing an Interface (implements)

- A class **implements** an interface with the `implements` keyword.
- It must implement **all** the abstract methods of the interface, or be `abstract` itself.
- Missing method: `Square is not abstract and does not override abstract method draw() in GeometricShape`.
- The implementing method must be `public`. If you write `void draw()`: `attempting to assign weaker access privileges; was public`.
- A variable of an interface type can point to an object of any class that implements it.

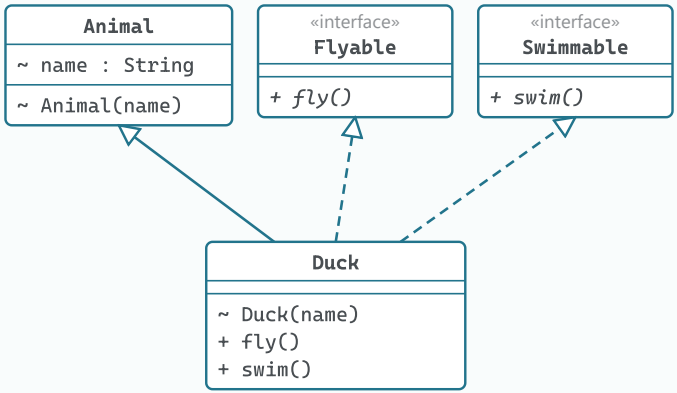
```
public class Square implements GeometricShape {
    // must implement draw
    public void draw() {
        System.out.println("Drawing a square");
    }
}
// in main:
GeometricShape g = new Square();
g.draw();
System.out.println(GeometricShape.SIDES);
```

Output

```
Drawing a square
4
```

One Class, Several Interfaces

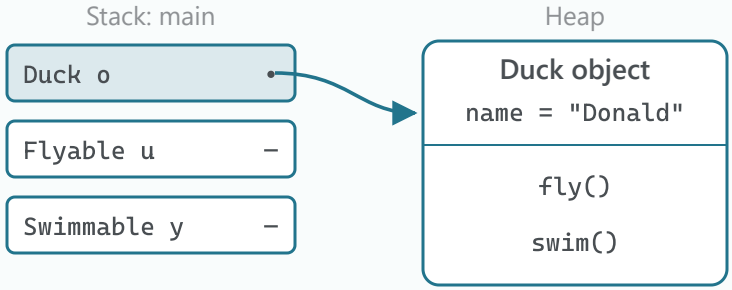
- A class can extend **at most one** superclass (`extends`)...
- ...but can implement **zero or more** interfaces (`implements A, B, …`).
- `extends` comes first, then `implements` .
- `class C extends A, B` → `error: '{' expected` .




```java
class Animal {
    String name;
    Animal(String name) { this.name = name; }
}
interface Flyable { void fly(); }
interface Swimmable { void swim(); }
class Duck extends Animal implements Flyable, Swimmable {
    Duck(String name) { super(name); }
    public void fly() { System.out.println(name + " is flying"); }
    public void swim() { System.out.println(name + " is swimming"); }
}
```

One Object, Different Interface References

```
▶ Duck o = new Duck("Donald");
2 Flyable u = o;
3 Swimmable y = o;
4 u.fly();
5 y.swim();
6 // u.swim();   compile error!
7 System.out.println(o.name);
8 System.out.println(u == o);
```



One Duck object is created on the heap; `o` points to it.

Step 1 / 8

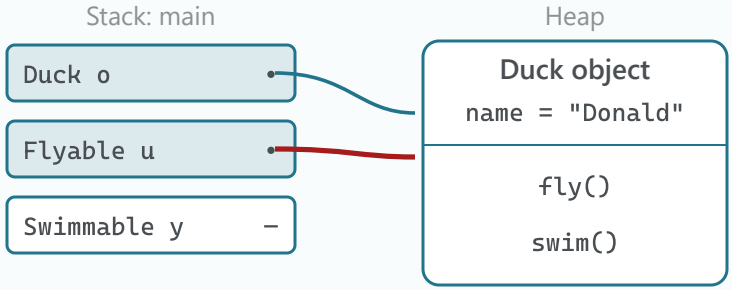
CALL STACK

`main()`

OUTPUT

One Object, Different Interface References

```
1 Duck o = new Duck("Donald");
▶ Flyable u = o;
3 Swimmable y = o;
4 u.fly();
5 y.swim();
6 // u.swim();   compile error!
7 System.out.println(o.name);
8 System.out.println(u == o);
```



Assignment does **not copy** the object: `u` points to the same object. A Duck is a Flyable → the conversion is automatic.

Step 2 / 8

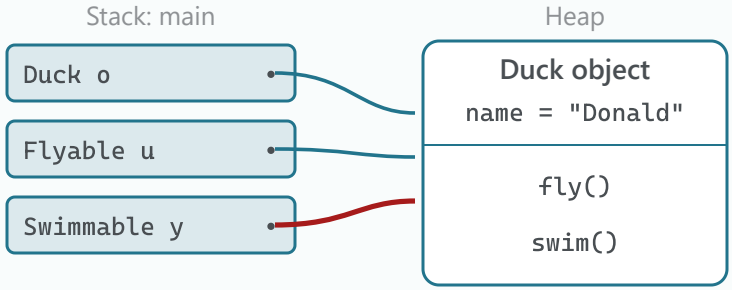
CALL STACK

`main()`

OUTPUT

One Object, Different Interface References

```
1 Duck o = new Duck("Donald");
2 Flyable u = o;
▶ Swimmable y = o;
4 u.fly();
5 y.swim();
6 // u.swim();   compile error!
7 System.out.println(o.name);
8 System.out.println(u == o);
```



`y` points to the same object too. One object, viewed through three different types.

Step 3 / 8

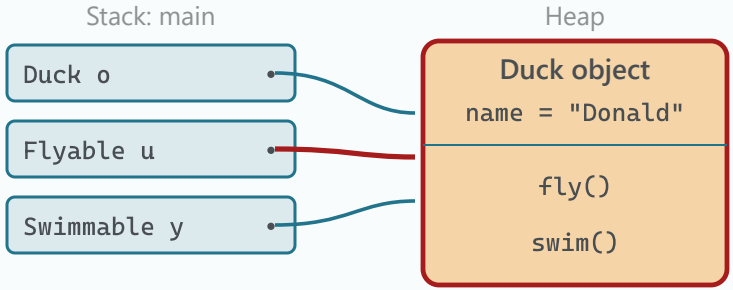
CALL STACK

`main()`

OUTPUT

One Object, Different Interface References

```
1 Duck o = new Duck("Donald");
2 Flyable u = o;
3 Swimmable y = o;
▶ u.fly();
5 y.swim();
6 // u.swim();   compile error!
7 System.out.println(o.name);
8 System.out.println(u == o);
```



`u.fly()` : `u` has type Flyable, so only `fly()` can be called on it. What runs is the object's own `fly()`.

Step 4 / 8

CALL STACK

`Duck.fly()`

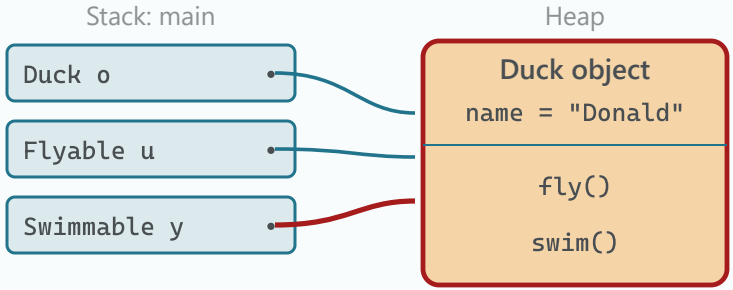
`main()`

OUTPUT

```
Donald is flying
```

One Object, Different Interface References

```
1 Duck o = new Duck("Donald");
2 Flyable u = o;
3 Swimmable y = o;
4 u.fly();
▶ y.swim();
6 // u.swim();   compile error!
7 System.out.println(o.name);
8 System.out.println(u == o);
```



`y.swim()` : type Swimmable knows `swim()` .

Step 5 / 8

CALL STACK

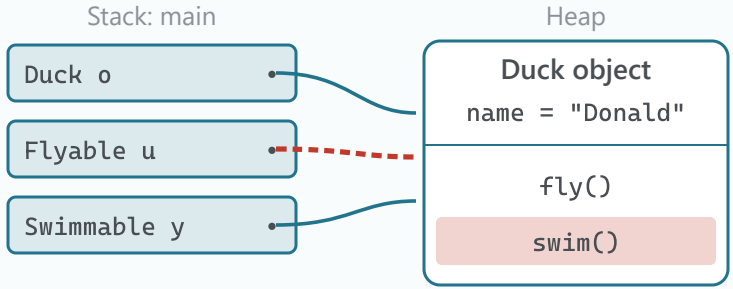
- **Duck.swim()**
- main()

OUTPUT

```
Donald is flying
Donald is swimming
```

One Object, Different Interface References

```
1 Duck o = new Duck("Donald");
2 Flyable u = o;
3 Swimmable y = o;
4 u.fly();
5 y.swim();
▶ // u.swim();   compile error!
7 System.out.println(o.name);
8 System.out.println(u == o);
```



Had we written `u.swim()` : `error: cannot find symbol` . The object has `swim()` , but the compiler looks at the **reference type**.

Step 6 / 8

CALL STACK

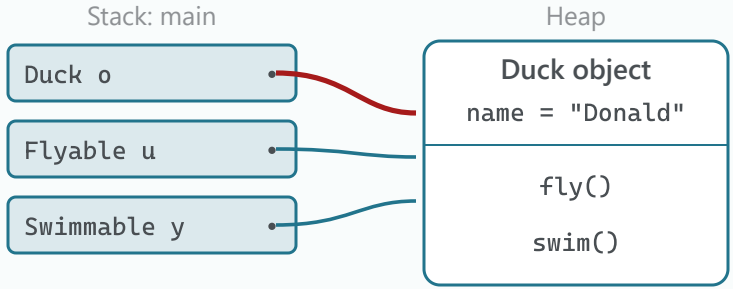
`main()`

OUTPUT

```
Donald is flying
Donald is swimming
```

One Object, Different Interface References

```
1 Duck o = new Duck("Donald");
2 Flyable u = o;
3 Swimmable y = o;
4 u.fly();
5 y.swim();
6 // u.swim();   compile error!
▶ System.out.println(o.name);
8 System.out.println(u == o);
```



`o` has type Duck: everything is accessible, including the field `name`.

CALL STACK

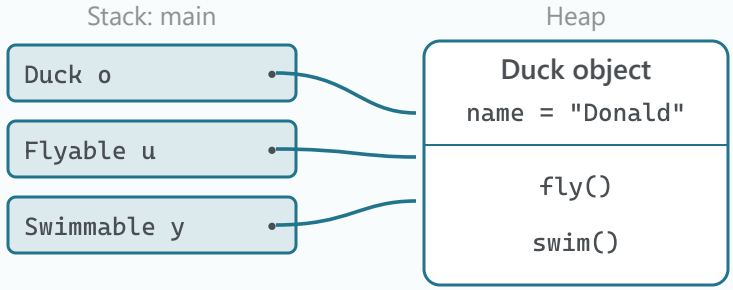
`main()`

OUTPUT

```
Donald is flying
Donald is swimming
Donald
```

One Object, Different Interface References

```
1 Duck o = new Duck("Donald");
2 Flyable u = o;
3 Swimmable y = o;
4 u.fly();
5 y.swim();
6 // u.swim();   compile error!
7 System.out.println(o.name);
▶ System.out.println(u == o);
```



`==` compares references: all point to the same object → `true`.

Step 8 / 8

CALL STACK

`main()`

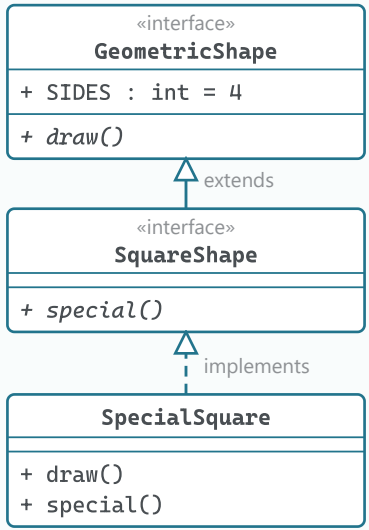
OUTPUT

```
Donald is flying
Donald is swimming
Donald
true
```

Interface Inheritance

An interface can extend another interface with `extends`, or **several** separated by commas: `interface C extends A, B`.

```
public interface SquareShape extends GeometricShape {
    // may add methods or constants
    void special();
}
class SpecialSquare implements SquareShape {
    public void draw() {
        System.out.println("Drawing a special square");
    }
    public void special() {
        System.out.println("Diagonal: " + Math.sqrt(2));
    }
}
// in main:
SquareShape k = new SpecialSquare();
k.draw();
k.special();
GeometricShape g = k;    // parent interface type
```

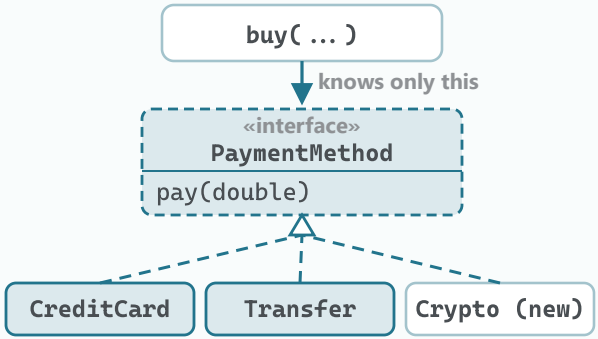


Output

```
Drawing a special square
Diagonal: 1.4142135623730951
```

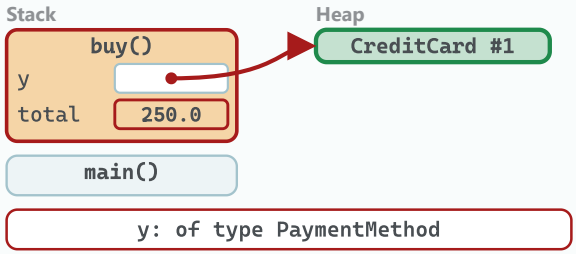
Programming to an Interface

- Client code depends on the **interface**, not on a concrete class: variables, parameters and return types use the interface.
- `buy(PaymentMethod y, double total)` does not know whether you pay by card or by transfer; it says `y.pay(total)` and the object does the rest.
- A new payment type (`Crypto`) just implements the interface; `buy` never changes (loose coupling).
- Testing gets easier: instead of a real bank, pass a **fake** class that implements the interface.
- The Java library works the same way: `List<String> l = new ArrayList<>();`



Payment Interface: Code Trace

```
 1 interface PaymentMethod { void pay(double amount); }
 2 class CreditCard implements PaymentMethod {
 3     public void pay(double amount) {
 4         System.out.println("Card: " + amount + " TL");
 5     }
 6 }
 7 class Transfer implements PaymentMethod {
 8     public void pay(double amount) {
 9         System.out.println("Transfer: " + (amount + 5) + " TL");
10     }
11 }
▶ static void buy(PaymentMethod y, double total) {
13     y.pay(total);
14 }
▶ buy(new CreditCard(), 250.0);   // main
16 buy(new Transfer(), 120.0);
```



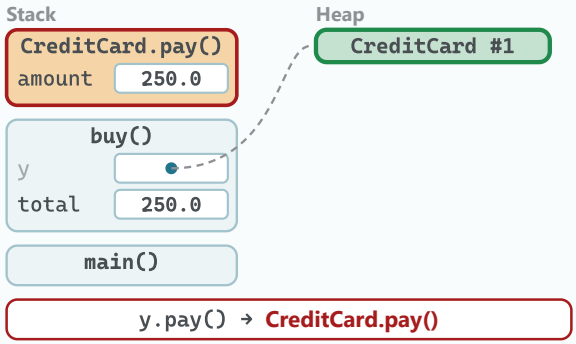
The parameter `y` has an interface type; the object passed in is a `CreditCard`.

Step 1 / 6

OUTPUT

Payment Interface: Code Trace

```
 1 interface PaymentMethod { void pay(double amount); }
 2 class CreditCard implements PaymentMethod {
 3     public void pay(double amount) {
 4         System.out.println("Card: " + amount + " TL");
 5     }
 6 }
 7 class Transfer implements PaymentMethod {
 8     public void pay(double amount) {
 9         System.out.println("Transfer: " + (amount + 5) + " TL");
10     }
11 }
12 static void buy(PaymentMethod y, double total) {
13     y.pay(total);
14 }
15 buy(new CreditCard(), 250.0);   // main
16 buy(new Transfer(), 120.0);
```



`y.pay(total)` is bound dynamically: the object is a card, so `CreditCard.pay()` runs.

Step 2 / 6

OUTPUT

```
Card: 250.0 TL
```

Payment Interface: Code Trace

```
 1 interface PaymentMethod { void pay(double amount); }
 2 class CreditCard implements PaymentMethod {
 3     public void pay(double amount) {
 4         System.out.println("Card: " + amount + " TL");
 5     }
 6 }
 7 class Transfer implements PaymentMethod {
 8     public void pay(double amount) {
 9         System.out.println("Transfer: " + (amount + 5) + " TL");
10     }
11 }
12 static void buy(PaymentMethod y, double total) {
13     y.pay(total);
14 }
▶ buy(new CreditCard(), 250.0);   // main
16 buy(new Transfer(), 120.0);
```



The call ends; nothing refers to the card object any more, so the garbage collector may reclaim it.

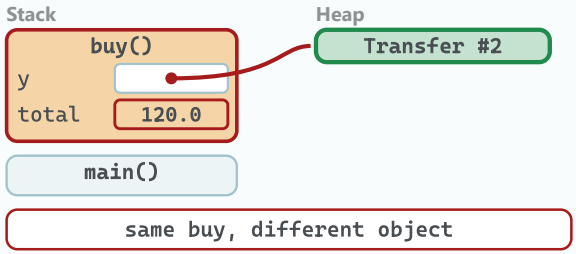
Step 3 / 6

OUTPUT

```
Card: 250.0 TL
```

Payment Interface: Code Trace

```
 1 interface PaymentMethod { void pay(double amount); }
 2 class CreditCard implements PaymentMethod {
 3     public void pay(double amount) {
 4         System.out.println("Card: " + amount + " TL");
 5     }
 6 }
 7 class Transfer implements PaymentMethod {
 8     public void pay(double amount) {
 9         System.out.println("Transfer: " + (amount + 5) + " TL");
10     }
11 }
▶ static void buy(PaymentMethod y, double total) {
13     y.pay(total);
14 }
15 buy(new CreditCard(), 250.0);   // main
▶ buy(new Transfer(), 120.0);
```



The same `buy`, this time with a `Transfer` object.

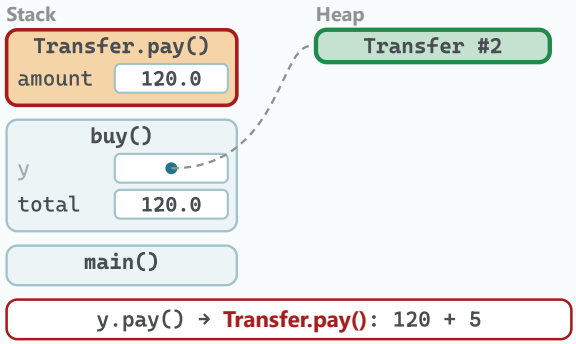
Step 4 / 6

OUTPUT

```
Card: 250.0 TL
```

Payment Interface: Code Trace

```
 1 interface PaymentMethod { void pay(double amount); }
 2 class CreditCard implements PaymentMethod {
 3     public void pay(double amount) {
 4         System.out.println("Card: " + amount + " TL");
 5     }
 6 }
 7 class Transfer implements PaymentMethod {
 8     public void pay(double amount) {
 9         System.out.println("Transfer: " + (amount + 5) + " TL");
10     }
11 }
12 static void buy(PaymentMethod y, double total) {
13     y.pay(total);
14 }
15 buy(new CreditCard(), 250.0);   // main
16 buy(new Transfer(), 120.0);
```



This time the same line runs `Transfer.pay()`, which adds a 5 TL fee: **125.0**.

Step 5 / 6

OUTPUT

```
Card: 250.0 TL
Transfer: 125.0 TL
```

Payment Interface: Code Trace

```
 1 interface PaymentMethod { void pay(double amount); }
 2 class CreditCard implements PaymentMethod {
 3     public void pay(double amount) {
 4         System.out.println("Card: " + amount + " TL");
 5     }
 6 }
 7 class Transfer implements PaymentMethod {
 8     public void pay(double amount) {
 9         System.out.println("Transfer: " + (amount + 5) + " TL");
10     }
11 }
▶ static void buy(PaymentMethod y, double total) {
▶     y.pay(total);
14 }
15 buy(new CreditCard(), 250.0);   // main
16 buy(new Transfer(), 120.0);
```

Stack

`main()`

Heap

`two behaviors, buy unchanged`

`buy` worked with two different classes without knowing either name; adding a new class would not change it.

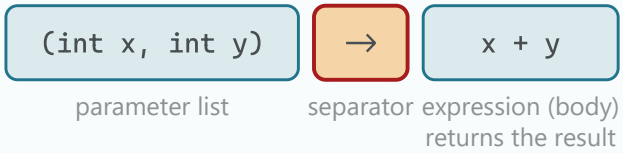
Step 6 / 6

OUTPUT

```
Card: 250.0 TL
Transfer: 125.0 TL
```

Anonymous (Lambda) Functions

- **Anonymous functions** are functions without a name: they stand directly for a piece of code.
- In Java they are called **lambda expressions** (**Java 8+**).
- Syntax: `(ParameterList) -> Expression`
- Multi-line body: `(a, b) -> { ...; return ...; }`
- Parameter types are usually left out and inferred: `(x, y) -> x + y`. With one parameter the parentheses can go too: `s -> s * 2`.



An addition lambda: `(int x, int y) -> x + y` takes two integers and adds them. The equivalent method:

```
static int add(int x, int y) {
    return x + y;
}
```

Functional Interfaces

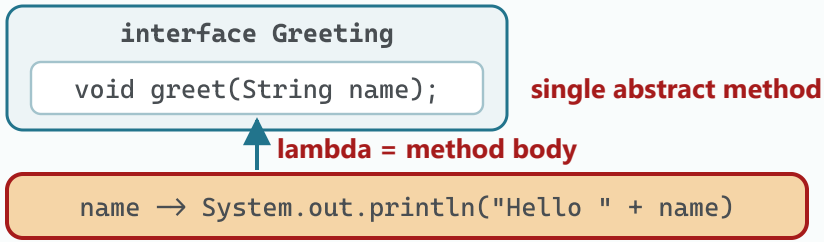
- **Functional interface**: an interface with **exactly one abstract method**. A lambda can only be assigned to such a type.
- The lambda supplies the body of that one method; parameter and return types are inferred from the method.
- `@FunctionalInterface` is optional; if present, the compiler refuses a second abstract method.
- A lambda does **not run** when it is defined; it runs when the method is called (`s.greet(...)`).

```
@FunctionalInterface
interface Greeting {
    void greet(String name);
}

Greeting s = name -> System.out.println("Hello " + name);
s.greet("Ayşe");
```

OUTPUT

```
Hello Ayşe
```



Built-in Functional Interfaces

You need not write an interface every time: package `java.util.function` provides the common shapes.

Interface	Input	Output	Method	Typical use
`Consumer<T>`	one value	none	`accept`	printing, saving
`Supplier<T>`	none	one value	`get`	producing a value
`Function<T, R>`	one value	one value	`apply`	transforming
`Predicate<T>`	one value	`boolean`	`test`	filtering, conditions

```
Consumer<String> print = m -> System.out.println(m);
print.accept("Consumer ran");
Supplier<Integer> dice = () -> 4;
System.out.println(dice.get());
Function<String, Integer> len = m -> m.length();
System.out.println(len.apply("Giresun"));
Predicate<Integer> isPositive = x -> x > 0;
System.out.println(isPositive.test(-7));
```

OUTPUT

```
Consumer ran
4
7
false
```

When Does a Lambda Run? Step by Step

```
▶ Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
4 for (int s : numbers) {
5     if (isEven.test(s)) {
6         result.add(s);
7     }
8 }
9 System.out.println(result);
```

`isEven: x → x % 2 == 0`

The lambda is stored as a `Predicate<Integer>` object; its body has **not run** yet. `isEven` is a reference to that behavior.

Step 1 / 14

OUTPUT

VARIABLES

`result = –`

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
▶ List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
4 for (int s : numbers) {
5     if (isEven.test(s)) {
6         result.add(s);
7     }
8 }
9 System.out.println(result);
```

isEven: x → x % 2 == 0

numbers | 3 | 8 | 5 | 12 |

The list: 3, 8, 5, 12.

Step 2 / 14

OUTPUT

VARIABLES

result = –

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
▶ List<Integer> result = new ArrayList<>();
4 for (int s : numbers) {
5     if (isEven.test(s)) {
6         result.add(s);
7     }
8 }
9 System.out.println(result);
```

`isEven: x → x % 2 == 0`

numbers: `3` `8` `5` `12`

result: empty

The result list starts empty.

Step 3 / 14

OUTPUT

VARIABLES

`result = []`

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
▶ for (int s : numbers) {
5     if (isEven.test(s)) {
6         result.add(s);
7     }
8 }
9 System.out.println(result);
```

`isEven: x → x % 2 == 0`

numbers: **3** | 8 | 5 | 12

result: empty

Next element: `s = 3`.

Step 4 / 14

OUTPUT

VARIABLES

`s = 3` `result = []`

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
4 for (int s : numbers) {
▶     if (isEven.test(s)) {
6         result.add(s);
7     }
8 }
9 System.out.println(result);
```

`isEven: x → x % 2 == 0` **test(3) → false**

numbers: **3** | 8 | 5 | 12

result: empty

`isEven.test(3)` runs the lambda body: `3 % 2 == 0` → **false**.

Step 5 / 14

OUTPUT

VARIABLES

`s = 3` `test = false` `result = []`

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
▶ for (int s : numbers) {
5     if (isEven.test(s)) {
6         result.add(s);
7     }
8 }
9 System.out.println(result);
```

`isEven: x → x % 2 == 0`

numbers: `3` `8` `5` `12`

result: empty

Next element: `s = 8`.

Step 6 / 14

OUTPUT

VARIABLES

`s = 8` `result = []`

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
4 for (int s : numbers) {
▶     if (isEven.test(s)) {
6         result.add(s);
7     }
8 }
9 System.out.println(result);
```

`isEven: x → x % 2 == 0` **test(8) → true**

numbers: [3] [8] [5] [12]

result: [empty]

`isEven.test(8)` runs the lambda body: `8 % 2 == 0` → **true**.

Step 7 / 14

OUTPUT

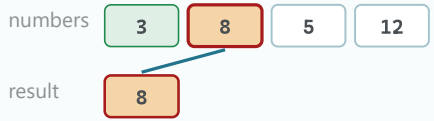
VARIABLES

`s = 8` `test = true` `result = []`

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
4 for (int s : numbers) {
5     if (isEven.test(s)) {
          result.add(s);
7     }
8 }
9 System.out.println(result);
```

`isEven: x → x % 2 == 0`



The condition holds: 8 is added to the result list.

Step 8 / 14

OUTPUT

VARIABLES

`s = 8` `test = true` `result = [8]`

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
▶ for (int s : numbers) {
5     if (isEven.test(s)) {
6         result.add(s);
7     }
8 }
9 System.out.println(result);
```

`isEven: x → x % 2 == 0`

numbers: 3 | 8 | 5 | 12

result: 8

Next element: `s = 5`.

Step 9 / 14

OUTPUT

VARIABLES

`s = 5` `result = [8]`

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
4 for (int s : numbers) {
▶     if (isEven.test(s)) {
6         result.add(s);
7     }
8 }
9 System.out.println(result);
```

`isEven: x → x % 2 == 0` **test(5) → false**

numbers: 3 | 8 | 5 | 12

result: 8

`isEven.test(5)` runs the lambda body: `5 % 2 == 0` → **false**.

Step 10 / 14

OUTPUT

VARIABLES

`s = 5` `test = false` `result = [8]`

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
▶ for (int s : numbers) {
5     if (isEven.test(s)) {
6         result.add(s);
7     }
8 }
9 System.out.println(result);
```

`isEven: x → x % 2 == 0`

numbers: `3` `8` `5` `12`

result: `8`

Next element: `s = 12`.

Step 11 / 14

OUTPUT

VARIABLES

`s = 12` `result = [8]`

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
4 for (int s : numbers) {
▶     if (isEven.test(s)) {
6         result.add(s);
7     }
8 }
9 System.out.println(result);
```

`isEven: x → x % 2 == 0` **test(12) → true**

numbers: 3 | 8 | 5 | 12

result: 8

`isEven.test(12)` runs the lambda body: `12 % 2 == 0` → **true**.

Step 12 / 14

OUTPUT

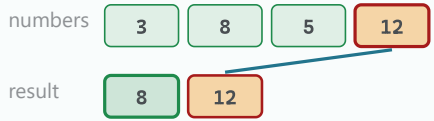
VARIABLES

`s = 12` `test = true` `result = [8]`

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
4 for (int s : numbers) {
5     if (isEven.test(s)) {
▶         result.add(s);
7     }
8 }
9 System.out.println(result);
```

`isEven: x → x % 2 == 0`



The condition holds: 12 is added to the result list.

Step 13 / 14

OUTPUT

VARIABLES

`s = 12` `test = true`

`result = [8, 12]`

When Does a Lambda Run? Step by Step

```
1 Predicate<Integer> isEven = x -> x % 2 == 0;
2 List<Integer> numbers = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> result = new ArrayList<>();
4 for (int s : numbers) {
5     if (isEven.test(s)) {
6         result.add(s);
7     }
8 }
▶ System.out.println(result);
```

`isEven: x → x % 2 == 0`

numbers: 3 | 8 | 5 | 12

result: 8 | 12

The loop ends and `[8, 12]` is printed. The lambda ran four times: once per `test` call.

Step 14 / 14

OUTPUT

```
[8, 12]
```

VARIABLES

`result = [8, 12]`

Lambda Example: Filtering a List

Inside `main`, with `import java.util.*;`, `java.util.function.IntBinaryOperator` and `java.util.stream.Collectors` imported:

```
IntBinaryOperator add = (int x, int y) -> x + y;
System.out.println(add.applyAsInt(5, 3));

List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
List<Integer> evens = numbers.stream().
                    filter(s -> s % 2 == 0).
                    collect(Collectors.toList());
System.out.println(evens);
System.out.println(numbers.stream().map(s -> s * s).toList());
```

OUTPUT

```
8
[2, 4, 6]
[1, 4, 9, 16, 25, 36, 49]
```

- The lambda is assigned to a **functional interface** type: `IntBinaryOperator` takes two `int`s and returns an `int`; the call is `applyAsInt`.
- `filter(s -> s % 2 == 0)` keeps the even numbers; `map` transforms.
- `.toList()` needs Java 16+; before that, `collect(Collectors.toList())`.

Lambdas vs. Anonymous Classes

```
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Anonymous class");
    }
};
Runnable r2 =
        () -> System.out.println("Lambda");
r1.run();
r2.run();
```

OUTPUT

```
Anonymous class
Lambda
```

Before lambdas, single-method behavior was written as an **anonymous class**. A lambda does the same in one line.

Limits

- A lambda can only be assigned to a **functional interface** type.
- Local variables it uses must be `final` or effectively final: never reassigned afterwards.
- Inside a lambda, `this` is the enclosing object; in an anonymous class it is the anonymous object itself.

```
int tax = 20;
Function<Integer, Integer> withTax =
        price -> price + price * tax / 100;
System.out.println(withTax.apply(150)); // 180
tax = 18;  // if added: compile error
```

```
error: local variables referenced from
  a lambda expression must be final
  or effectively final
```

Where Lambdas Are Used

- **Collection operations:** filtering, sorting and mapping lists (`filter` , `sort` , `map` , `forEach`).
- **Threads:** concisely supplying the work to run in parallel.
- **More readable code:** instead of defining a separate function for a small one-off operation.

```
List<String> names = new ArrayList<>(List.of("Zeynep", "Ali", "Mehmet"));
names.sort((a, b) -> a.length() - b.length());
System.out.println(names);
names.forEach(name -> System.out.println("Hello " + name));
Thread t = new Thread(() -> System.out.println("Thread ran"));
t.start();
```

OUTPUT

```
[Ali, Zeynep, Mehmet]
Hello Ali
Hello Zeynep
Hello Mehmet
Thread ran
```

The sort key is a lambda: a negative difference puts `a` first. Names of equal length (Zeynep, Mehmet) keep their original order. `() -> ...` is a lambda with no parameters.

Default Methods (default, Java 8+)

- An interface can have default (`default`) methods **with a body**.
- Implementing classes **inherit** them and may override them. The goal: add a method to an interface without **breaking** the old classes that implement it.

```
interface Collection {
    void add(Object o);
    void remove(Object o);
    int size();
    default boolean isEmpty() {
        return size() == 0;
    }
}
```

```
class Cart implements Collection {
    private int count = 0;
    public void add(Object o) { count++; }
    public void remove(Object o) { count--; }
    public int size() { return count; }
    // no isEmpty(): the default version is used
}
// in main:
Cart cart = new Cart();
System.out.println(cart.isEmpty());
cart.add("apple");
cart.add("cherry");
System.out.println(cart.size() + " " + cart.isEmpty());
```

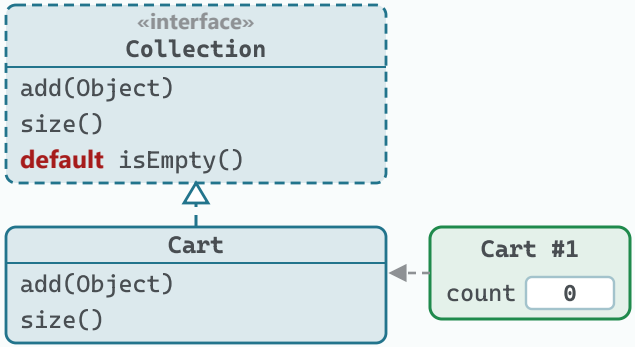
Output

```
true
2 false
```

`isEmpty()` calls the abstract `size()`; which `size()` runs depends on the object. (This `Collection` is our own interface, not `java.util.Collection`.)

Default Method: Code Trace

```
 1 interface Collection {
 2     void add(Object o);
 3     int size();
 4     default boolean isEmpty() {
 5         return size() == 0;
 6     }
 7 }
 8 class Cart implements Collection {
 9     private int count = 0;
10     public void add(Object o) { count++; }
11     public int size() { return count; }
12 }
▶ Collection c = new Cart();      // main
14 System.out.println(c.isEmpty());
15 c.add("apple");
16 System.out.println(c.isEmpty());
```



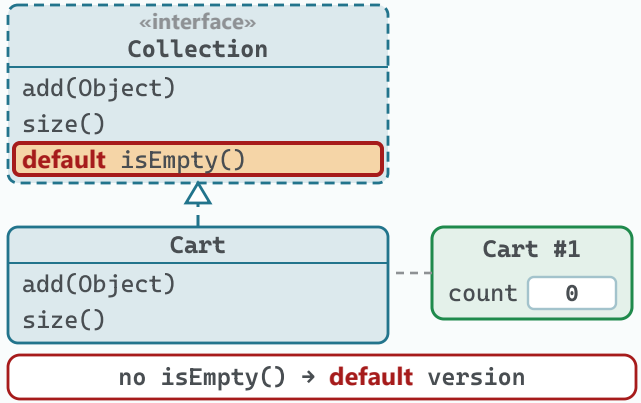
`Cart` wrote only the two abstract methods, not `isEmpty()`.

Step 1 / 6

OUTPUT

Default Method: Code Trace

```
 1 interface Collection {
 2     void add(Object o);
 3     int size();
 ▶     default boolean isEmpty() {
 5         return size() == 0;
 6     }
 7 }
 8 class Cart implements Collection {
 9     private int count = 0;
10     public void add(Object o) { count++; }
11     public int size() { return count; }
12 }
13 Collection c = new Cart();       // main
 ▶ System.out.println(c.isEmpty());
15 c.add("apple");
16 System.out.println(c.isEmpty());
```



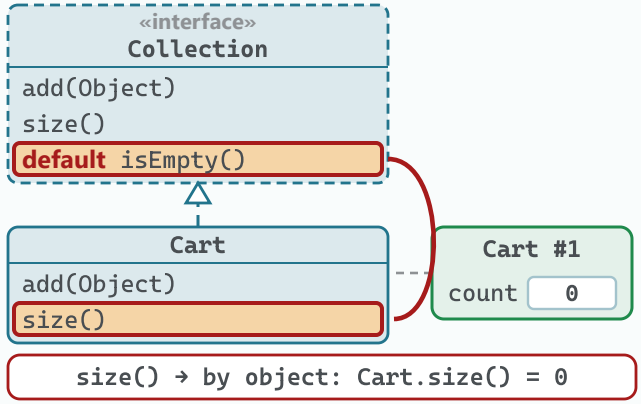
Lookup finds no `isEmpty()` in `Cart`, so the interface's **default** body runs.

Step 2 / 6

OUTPUT

Default Method: Code Trace

```
1  interface Collection {
2      void add(Object o);
3      int size();
4      default boolean isEmpty() {
5          return size() == 0;
6      }
7  }
8  class Cart implements Collection {
9      private int count = 0;
10     public void add(Object o) { count++; }
11     public int size() { return count; }
12 }
13 Collection c = new Cart();       // main
14 System.out.println(c.isEmpty());
15 c.add("apple");
16 System.out.println(c.isEmpty());
```



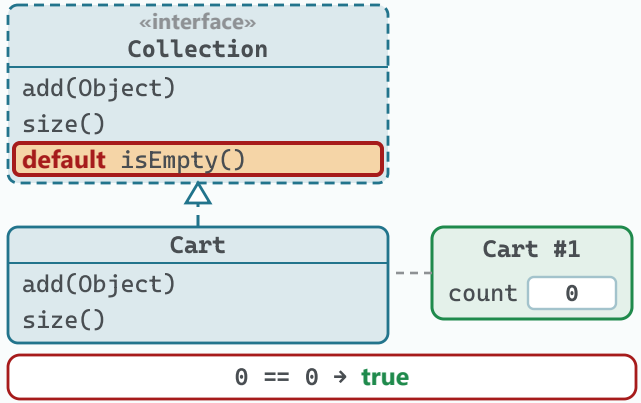
The default method calls the abstract `size()`; the **object** decides which body runs: `Cart.size()` → 0.

Step 3 / 6

OUTPUT

Default Method: Code Trace

```
1  interface Collection {
2      void add(Object o);
3      int size();
4      default boolean isEmpty() {
▶          return size() == 0;
6      }
7  }
8  class Cart implements Collection {
9      private int count = 0;
10     public void add(Object o) { count++; }
11     public int size() { return count; }
12 }
13 Collection c = new Cart();       // main
14 System.out.println(c.isEmpty());
15 c.add("apple");
16 System.out.println(c.isEmpty());
```



`0 == 0` : **true**.

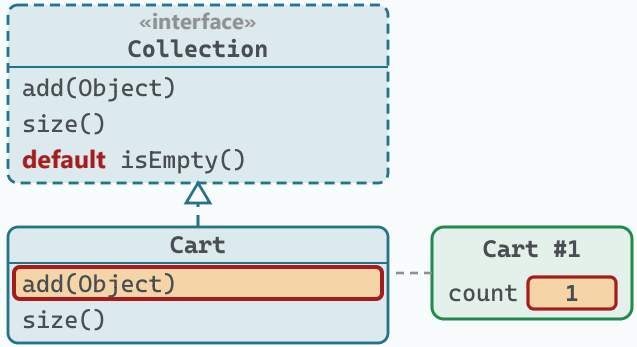
Step 4 / 6

OUTPUT

```
true
```

Default Method: Code Trace

```
 1 interface Collection {
 2     void add(Object o);
 3     int size();
 4     default boolean isEmpty() {
 5         return size() == 0;
 6     }
 7 }
 8 class Cart implements Collection {
 9     private int count = 0;
▶    public void add(Object o) { count++; }
11     public int size() { return count; }
12 }
13 Collection c = new Cart();       // main
14 System.out.println(c.isEmpty());
▶  c.add("apple");
16 System.out.println(c.isEmpty());
```



`add` is Cart's version: `count` becomes 1.

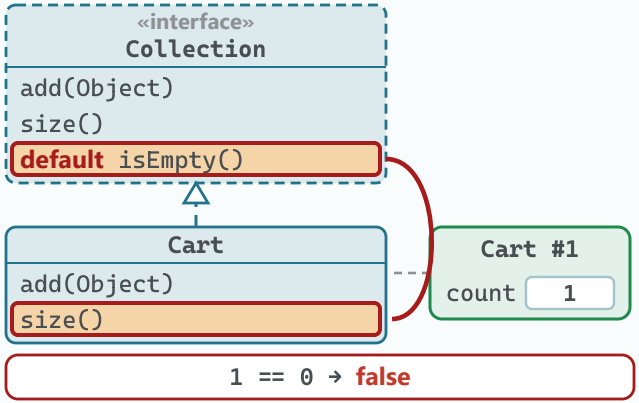
Step 5 / 6

OUTPUT

```
true
```

Default Method: Code Trace

```
 1 interface Collection {
 2     void add(Object o);
 3     int size();
 4     default boolean isEmpty() {
 ▶         return size() == 0;
 6     }
 7 }
 8 class Cart implements Collection {
 9     private int count = 0;
10     public void add(Object o) { count++; }
 ▶     public int size() { return count; }
12 }
13 Collection c = new Cart();       // main
14 System.out.println(c.isEmpty());
15 c.add("apple");
 ▶ System.out.println(c.isEmpty());
```



The same default body now sees 1: **false**. A default method added after the interface was written reaches old classes without breaking them.

Step 6 / 6

OUTPUT

```
true
false
```

static Methods (Java 8+)

```
interface Geometry {
    static double circleArea(double r) { return Math.PI * r * r; }
    static double squareArea(double s) { return s * s; }
    static double perimeter(double s) { return s * 4.0; }
    static double boxVolume(double w, double h, double d) {
        return w * h * d;
    }
}
// in main:
System.out.println(Geometry.circleArea(2));
System.out.println(Geometry.squareArea(3));
System.out.println(Geometry.perimeter(2.5));
System.out.println(Geometry.boxVolume(2, 3, 4));
```

Output

```
12.566370614359172
9.0
10.0
24.0
```

- **Helper** functions not tied to an object; called only **through the interface name**: `Geometry.circleArea(2)`.
- They are **not inherited** by implementing classes: even with `Circle implements Geometry`, `Circle.squareArea(2)` → `cannot find symbol`.
- A bare `PI` gives `cannot find symbol`; the constant lives in `Math`: `Math.PI`.

Default Method Conflicts

- If two interfaces bring default methods with the **same signature**, the class cannot tell which to take; it **must write the method itself** (override it).
- A particular interface's version can be called with `InterfaceName.super.method()`.

```
interface Greeter {
    default void greet() { System.out.println("Hello"); }
}
interface Welcomer {
    default void greet() { System.out.println("Welcome"); }
}
class Clerk implements Greeter, Welcomer {
    public void greet() {       // must override
        Greeter.super.greet();
        Welcomer.super.greet();
    }
}
```

`new Clerk().greet();`

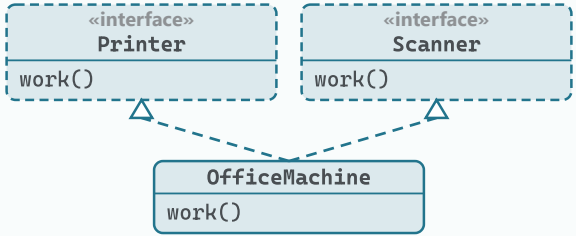
```
Hello
Welcome
```

Without `greet()`

```
error: types Greeter and Welcomer are incompatible;
  class Clerk inherits unrelated defaults for greet()
  from types Greeter and Welcomer
```

Default Conflict: Code Trace

```
1  interface Printer {
2      default void work() { System.out.println("Printing"); }
3  }
4  interface Scanner {
5      default void work() { System.out.println("Scanning"); }
6  }
7  class OfficeMachine implements Printer, Scanner {
8      public void work() {
9          Scanner.super.work();
10         Printer.super.work();
11         System.out.println("Copy ready");
12     }
13 }
▶  Printer y = new OfficeMachine();   // main
15 y.work();
```



conflict → the class **writes** it itself

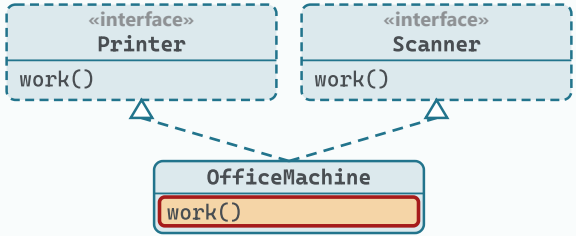
Both interfaces bring a default body for `work()`. To resolve the ambiguity, `OfficeMachine` writes the method itself.

Step 1 / 6

OUTPUT

Default Conflict: Code Trace

```
1  interface Printer {
2      default void work() { System.out.println("Printing"); }
3  }
4  interface Scanner {
5      default void work() { System.out.println("Scanning"); }
6  }
7  class OfficeMachine implements Printer, Scanner {
▶      public void work() {
9          Scanner.super.work();
10         Printer.super.work();
11         System.out.println("Copy ready");
12     }
13 }
14 Printer y = new OfficeMachine();   // main
▶ y.work();
```



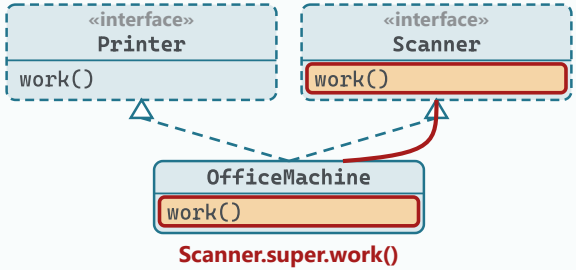
The reference type is `Printer`, but the object is an `OfficeMachine`: the class's own `work()` runs.

Step 2 / 6

OUTPUT

Default Conflict: Code Trace

```
1  interface Printer {
2      default void work() { System.out.println("Printing"); }
3  }
4  interface Scanner {
▶      default void work() { System.out.println("Scanning"); }
6  }
7  class OfficeMachine implements Printer, Scanner {
8      public void work() {
▶          Scanner.super.work();
10         Printer.super.work();
11         System.out.println("Copy ready");
12     }
13 }
14 Printer y = new OfficeMachine();   // main
15 y.work();
```



`Scanner.super.work()` : calls the default body of the named interface.

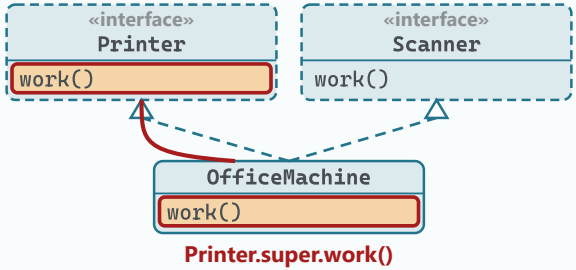
Step 3 / 6

OUTPUT

```
Scanning
```

Default Conflict: Code Trace

```
1  interface Printer {
2      default void work() { System.out.println("Printing"); }
3  }
4  interface Scanner {
5      default void work() { System.out.println("Scanning"); }
6  }
7  class OfficeMachine implements Printer, Scanner {
8      public void work() {
9          Scanner.super.work();
10         Printer.super.work();
11         System.out.println("Copy ready");
12     }
13 }
14 Printer y = new OfficeMachine();   // main
15 y.work();
```



Then `Printer`'s. The class chooses which versions to use and in what order.

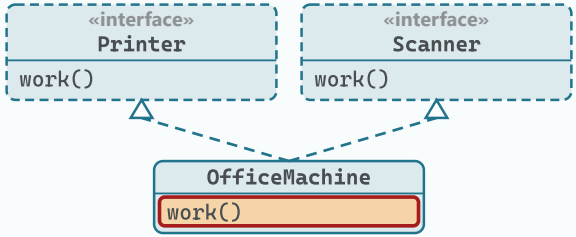
Step 4 / 6

OUTPUT

```
Scanning
Printing
```

Default Conflict: Code Trace

```
1  interface Printer {
2      default void work() { System.out.println("Printing"); }
3  }
4  interface Scanner {
5      default void work() { System.out.println("Scanning"); }
6  }
7  class OfficeMachine implements Printer, Scanner {
8      public void work() {
9          Scanner.super.work();
10         Printer.super.work();
▶          System.out.println("Copy ready");
12     }
13 }
14 Printer y = new OfficeMachine();   // main
15 y.work();
```



Finally the class adds its own step: behavior that combines both capabilities.

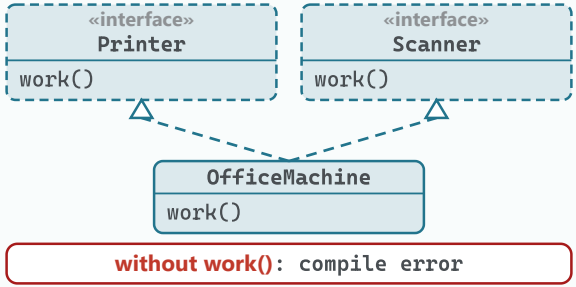
Step 5 / 6

OUTPUT

```
Scanning
Printing
Copy ready
```

Default Conflict: Code Trace

```
1  interface Printer {
2      default void work() { System.out.println("Printing"); }
3  }
4  interface Scanner {
5      default void work() { System.out.println("Scanning"); }
6  }
7  class OfficeMachine implements Printer, Scanner {
8      public void work() {
9          Scanner.super.work();
10         Printer.super.work();
11         System.out.println("Copy ready");
12     }
13 }
14 Printer y = new OfficeMachine();   // main
▶  y.work();
```



without work(): compile error

Had `OfficeMachine` not written `work()`, the class would not compile: `inherits unrelated defaults`.

Step 6 / 6

OUTPUT

```
Scanning
Printing
Copy ready
```

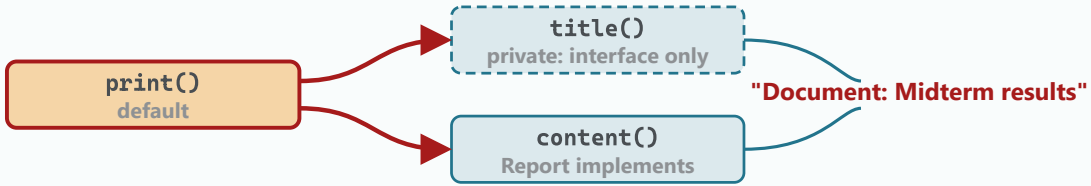
Private Interface Methods (Java 9+)

```
interface Printable {
    String content();
    default void print() {
        System.out.println(title() + ": " + content());
    }
    private String title() {   // Java 9+
        return "Document";
    }
}
class Report implements Printable {
    public String content() { return "Midterm results"; }
}
```

```
new Report().print();
```

```
Document: Midterm results
```

- `private` methods let default (and static) methods share **common helper code** without exposing it.
- They have bodies and can be called only from **inside** the interface.
- Implementing classes **cannot see** them: `title()` cannot be called inside `Report`.
- Flow: `print()` (default) → `title()` (private) + `content()` (the abstract method Report implements).



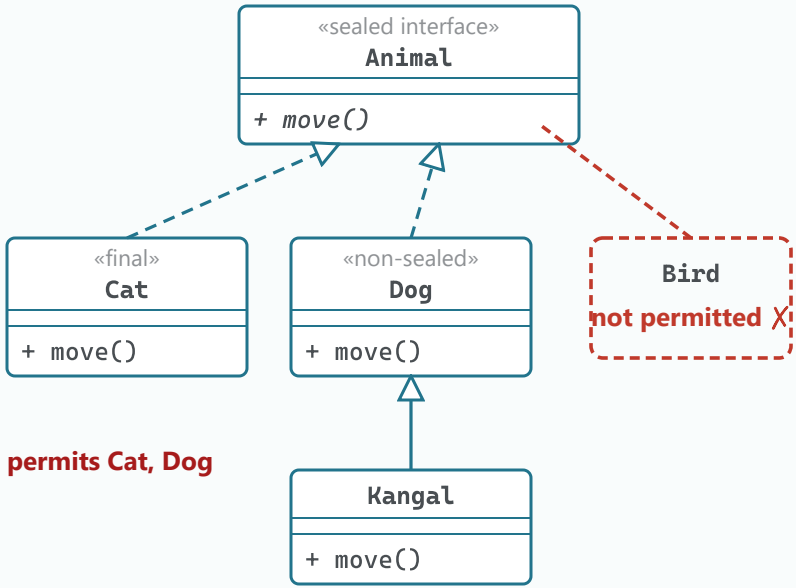
Interface Members: Summary

Member	Example	Implicit modifiers	Body	Java
Abstract method	`void draw();`	`public abstract`	none	1.0
Constant	`int SIDES = 4;`	`public static final`	—	1.0
Default method	`default boolean isEmpty() {…}`	`public`	yes	8
static method	`static double squareArea(…) {…}`	`public`	yes	8
Private method	`private String title() {…}`	inside the interface only	yes	9
Instance field, constructor	—	**Not allowed**: an interface has no instances and holds no state		

The rule for an implementing class: write every **abstract** method of the interface as `public`. Default methods are optional; static and private methods are not inherited anyway.

Sealing: `sealed`, `non-sealed`, `permits`

- Classes and interfaces can be **sealed** (Java 17+).
- `sealed` : **only the permitted** subtypes may extend or implement the type.
- `permits` : the list of permitted subtypes.
- Each permitted subtype **must** pick one of:
  `final` : the hierarchy ends here;
  `sealed` : restricts again;
  `non-sealed` : lifts the restriction, anyone may extend it.
- Benefit: the compiler **knows** every subtype (a closed hierarchy).



Sealed Interface Example

```
public sealed interface Animal permits Cat, Dog {
    void move();
}
final class Cat implements Animal {
    public void move() {
        System.out.println("Cat walks silently");
    }
}
non-sealed class Dog implements Animal {
    public void move() {
        System.out.println("Dog runs");
    }
}
class Kangal extends Dog {   // open
    public void move() {
        System.out.println("Kangal guards the flock");
    }
}
```

```
// Java 21: switch covers all subtypes,
// no default branch needed
static String kind(Animal h) {
    return switch (h) {
        case Cat k -> "cat";
        case Dog k -> "dog";
    };
}
// in main:
Animal[] animals =
    { new Cat(), new Dog(), new Kangal() };
for (Animal h : animals) {
    h.move();
    System.out.println(kind(h));
}
```

Output

```
Cat walks silently
cat
Dog runs
dog
Kangal guards the flock
dog
```

COMPARISON

Interfaces vs. Abstract Classes

Aspect	Interface	Abstract class
Declare / use	`interface` · `implements`	`abstract class` · `extends`
Methods	Abstract; Java 8+ `default` and `static`, Java 9+ `private`	Both abstract and concrete methods
Fields	Only `public static final` constants; **no** instance fields	Any fields (instance fields included)
Constructor	None	Allowed; a subclass calls it with `super(…)`
Access	Members implicitly `public` (except private helpers)	`public`, `protected`, `private`, package
Inheritance	A class implements **many** interfaces; an interface extends several interfaces → a form of multiple inheritance	A class extends exactly **one** class
Purpose	A shared capability for unrelated classes (“can do”)	Grouping classes that share fields (“is a kind of”)
Objects	Neither can be instantiated with `new`; both are used as reference types	

COMPARISON

Which One to Choose?

Use an abstract class when

- The subclasses are **close relatives**: Square and Circle are both Shapes.
- You need shared **fields** and a **constructor** (`name`).
- Part of the common code is ready (template method).

Use an interface when

- **Unrelated** classes share a capability: Bird, Plane and Drone are all Flyable.
- A class needs **several** capabilities.
- You want a **contract** (API) between client code and implementing code.

Often the two are used **together**:

```
abstract class Animal implements Noisy { … }
class Duck extends Animal implements Flyable, Swimmable { … }
```

Java forbids multiple inheritance of classes: if two superclasses define the same field or method differently, it is unclear which one applies (the diamond problem). Interfaces carry no state, so the problem largely disappears; the compiler makes the class resolve any default conflict.

Why Generics?

With Object

```
class ObjBox {
    private Object content;
    void put(Object o) { content = o; }
    Object get() { return content; }
}
ObjBox k = new ObjBox();
k.put("Hello");
String s = (String) k.get(); // cast required
k.put(42);                   // no complaint
String t = (String) k.get(); // runtime error
```

```
Exception in thread "main"
java.lang.ClassCastException
```

With a generic class

```
class Box<T> {
    private T content;
    void put(T x) { content = x; }
    T get() { return content; }
}
Box<String> k = new Box<>();
k.put("Hello");
String s = k.get();          // no cast
k.put(42);                   // won't compile
```

```
error: incompatible types:
  int cannot be converted to String
```

- **Type safety**: a value of the wrong type is caught at **compile time**, not at run time.
- No explicit cast when reading; one implementation works for every type.

Generic Classes and Generic Methods

```
class Box<T> {                    // T: type parameter
    private T content;
    void put(T x) { content = x; }
    T get() { return content; }
}
static <T> void print(T item) {    // generic method
    System.out.println("Item: " + item);
}
// in main:
Box<String> k1 = new Box<>();
k1.put("Hello");
Box<Integer> k2 = new Box<>();
k2.put(42);
int n = k2.get() + 1;               // unboxing
System.out.println(k1.get() + " " + n);
print(3.14);
print('A');
```

Output

```
Hello 43
Item: 3.14
Item: A
```

- `Box<String>` : the type argument takes the place of `T` .
- `new Box<>()` : the diamond (`<>`) infers the type from the left side.
- A generic method declares its type parameter before the return type: `<T>` ; the type is inferred from the argument.
- The type argument must be a reference type: `Box<int>` does not compile; write `Box<Integer>` .
- Common names: `T` type, `E` element, `K` key, `V` value, `R` result.

Bounded Type Parameters

`<T extends B>` : `T` can only be `B` or a subtype of it (an upper bound). In return, the method body may use `B` 's methods.

```
static <T extends Number> double twice(T x) {
    return x.doubleValue() * 2;   // a Number method
}
// in main:
System.out.println(twice(21));    // T = Integer
System.out.println(twice(1.5));   // T = Double
System.out.println(twice("ten")); // won't compile
```

First two lines

```
42.0
3.0
```

Third line

```
error: method twice in class Main cannot be
  applied to given types; ... inference variable T
  has incompatible bounds (upper: Number, lower: String)
```

An interface can be a bound too (still written with `extends`). The method below finds the largest element of any **comparable** type (`Integer` , `String` ...), since it is guaranteed to be able to call `compareTo` :

```
static <T extends Comparable<T>> T max(T[] arr)
```

`max`: Code Trace

```
1  static <T extends Comparable<T>> T max(T[] arr) {
2      T best = arr[0];
3      for (int i = 1; i < arr.length; i++) {
4          if (arr[i].compareTo(best) > 0) {
5              best = arr[i];
6          }
7      }
8      return best;
9  }
10 // in main:
▶  Integer[] numbers = {7, 12, 5};
▶  String[] fruits = {"apple", "pear", "cherry"};
13 System.out.println(max(numbers));
14 System.out.println(max(fruits));
```

two arrays

7	12	5

best ?

Both arrays qualify: `Integer` and `String` implement `Comparable` with their own type. The numbers are autoboxed.

Step 1 / 9

OUTPUT

`max`: Code Trace

```
▶ static <T extends Comparable<T>> T max(T[] arr) {
▶     T best = arr[0];
3     for (int i = 1; i < arr.length; i++) {
4         if (arr[i].compareTo(best) > 0) {
5             best = arr[i];
6         }
7     }
8     return best;
9 }
10 // in main:
11 Integer[] numbers = {7, 12, 5};
12 String[] fruits = {"apple", "pear", "cherry"};
▶ System.out.println(max(numbers));
14 System.out.println(max(fruits));
```

T = Integer





The compiler infers `T` as `Integer`. The first element is the first candidate: `best = 7`.

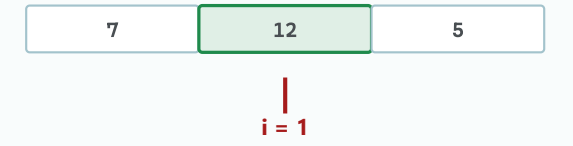
Step 2 / 9

OUTPUT

`max`: Code Trace

```
1  static <T extends Comparable<T>> T max(T[] arr) {
2      T best = arr[0];
▶      for (int i = 1; i < arr.length; i++) {
▶          if (arr[i].compareTo(best) > 0) {
▶              best = arr[i];
6          }
7      }
8      return best;
9  }
10 // in main:
11 Integer[] numbers = {7, 12, 5};
12 String[] fruits = {"apple", "pear", "cherry"};
13 System.out.println(max(numbers));
14 System.out.println(max(fruits));
```

T = Integer



`compareTo = 1 > 0 → best = 12`

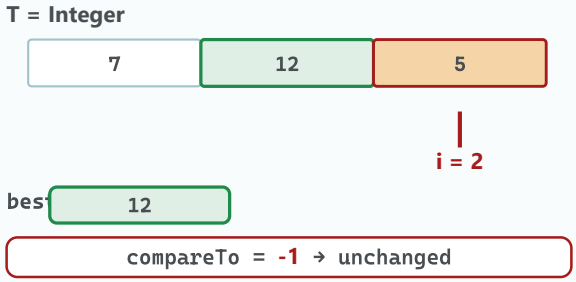
A positive `compareTo` means the element is larger: `best = 12`.

Step 3 / 9

OUTPUT

`max`: Code Trace

```
1  static <T extends Comparable<T>> T max(T[] arr) {
2      T best = arr[0];
▶      for (int i = 1; i < arr.length; i++) {
▶          if (arr[i].compareTo(best) > 0) {
5              best = arr[i];
6          }
7      }
8      return best;
9  }
10 // in main:
11 Integer[] numbers = {7, 12, 5};
12 String[] fruits = {"apple", "pear", "cherry"};
13 System.out.println(max(numbers));
14 System.out.println(max(fruits));
```



`compareTo = -1 → unchanged`

Negative result: 5 is smaller, so `best` stays.

Step 4 / 9

OUTPUT

`max`: Code Trace

```
1  static <T extends Comparable<T>> T max(T[] arr) {
2      T best = arr[0];
3      for (int i = 1; i < arr.length; i++) {
4          if (arr[i].compareTo(best) > 0) {
5              best = arr[i];
6          }
7      }
▶      return best;
9  }
10 // in main:
11 Integer[] numbers = {7, 12, 5};
12 String[] fruits = {"apple", "pear", "cherry"};
▶  System.out.println(max(numbers));
14 System.out.println(max(fruits));
```

T = Integer





The loop ends and **12** is returned. The return type is `T` = `Integer`: no cast needed.

Step 5 / 9

OUTPUT

```
12
```

`max`: Code Trace

```
1  static <T extends Comparable<T>> T max(T[] arr) {
▶      T best = arr[0];
3      for (int i = 1; i < arr.length; i++) {
4          if (arr[i].compareTo(best) > 0) {
5              best = arr[i];
6          }
7      }
8      return best;
9  }
10 // in main:
11 Integer[] numbers = {7, 12, 5};
12 String[] fruits = {"apple", "pear", "cherry"};
13 System.out.println(max(numbers));
▶  System.out.println(max(fruits));
```

T = String





The same method, now with `T = String`. `compareTo` now compares in lexicographic order.

Step 6 / 9

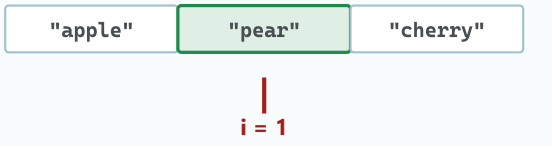
OUTPUT

```
12
```

`max`: Code Trace

```
1  static <T extends Comparable<T>> T max(T[] arr) {
2      T best = arr[0];
▶      for (int i = 1; i < arr.length; i++) {
▶          if (arr[i].compareTo(best) > 0) {
▶              best = arr[i];
6          }
7      }
8      return best;
9  }
10 // in main:
11 Integer[] numbers = {7, 12, 5};
12 String[] fruits = {"apple", "pear", "cherry"};
13 System.out.println(max(numbers));
14 System.out.println(max(fruits));
```

T = String



`compareTo = 6 > 0 → best = "pear"`

The first differing letters give `'p' − 'a'` = 15: positive, so `best = "pear"`.

Step 7 / 9

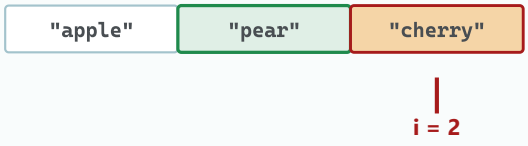
OUTPUT

```
12
```

`max`: Code Trace

```
1  static <T extends Comparable<T>> T max(T[] arr) {
2      T best = arr[0];
▶      for (int i = 1; i < arr.length; i++) {
▶          if (arr[i].compareTo(best) > 0) {
5              best = arr[i];
6          }
7      }
8      return best;
9  }
10 // in main:
11 Integer[] numbers = {7, 12, 5};
12 String[] fruits = {"apple", "pear", "cherry"};
13 System.out.println(max(numbers));
14 System.out.println(max(fruits));
```

T = String



`compareTo = -10 → unchanged`

`'c' - 'p'` = −13: negative, `best` stays.

Step 8 / 9

OUTPUT

```
12
```

`max`: Code Trace

```
1  static <T extends Comparable<T>> T max(T[] arr) {
2      T best = arr[0];
3      for (int i = 1; i < arr.length; i++) {
4          if (arr[i].compareTo(best) > 0) {
5              best = arr[i];
6          }
7      }
▶      return best;
9  }
10 // in main:
11 Integer[] numbers = {7, 12, 5};
12 String[] fruits = {"apple", "pear", "cherry"};
13 System.out.println(max(numbers));
▶  System.out.println(max(fruits));
```

T = String





pear is returned. One implementation worked for two types; thanks to the bound, the `compareTo` call is checked at compile time.

Step 9 / 9

OUTPUT

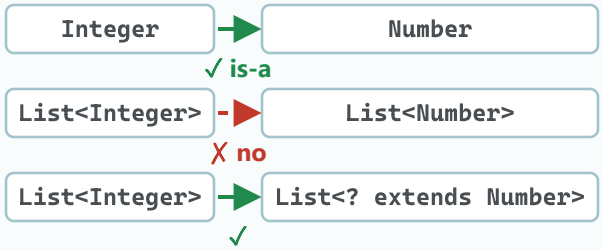
```
12
pear
```

Wildcards (`?`) and Invariance

```java
static double sum(List<? extends Number> list) {
    double t = 0;
    for (Number n : list) t += n.doubleValue();  // read
    return t;
}
static void fill(List<? super Integer> list) {
    list.add(1);                                 // write
    list.add(2);
}
// in main:
List<Integer> ints = Arrays.asList(1, 2, 3);
List<Number> numbers = new ArrayList<>();
fill(numbers);
numbers.add(2.5);
System.out.println(sum(ints) + " " + numbers);
// List<Number> x = ints;   error
```

Output

```
6.0 [1, 2, 2.5]
```



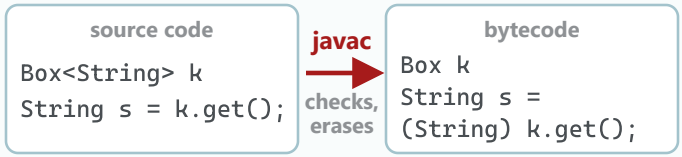
- **Invariance**: `List<Integer>` is not a subtype of `List<Number>`; if it were, you could add `2.5` to the list.
- `?`: some unknown type.
- `? extends T`: the list **produces** values; you read from it.
- `? super T`: the list **consumes** values; you write to it.
- Rule (PECS): *Producer Extends, Consumer Super*.

Type Erasure

```
class Box<T> {
    private T content;
    // T make() { return new T(); }   error
}
// in main:
Box<String> a = new Box<>();
Box<Integer> b = new Box<>();
System.out.println(a.getClass() == b.getClass());
System.out.println(a.getClass().getName());
```

Output

```
true
Box
```



- Generic type information is mostly for the **compiler**; after checking it is erased (*type erasure*). `T` becomes `Object` (or its bound).
- At run time `Box<String>` and `Box<Integer>` are **the same class**.
- That is why you cannot write `new T()` (`unexpected type`) or `new T[10]` (`generic array creation`).

Annotations

An *annotation* is **metadata** attached to a class, method, field or parameter. It changes no behavior by itself; the compiler, tools or the running program (via reflection) read it.

Annotation	Read by	Effect
`@Override`	compiler	Error if the supertype has no such method
`@Deprecated`	compiler	Obsolete element; code using it gets a warning
`@SuppressWarnings("unchecked")`	compiler	Silences the named warning
`@FunctionalInterface`	compiler	Checks that the interface has exactly one abstract method
`@Test` (JUnit)	test tool	Finds and runs test methods

```
@FunctionalInterface
interface Operation {
    int apply(int a, int b);
    int negate(int a);    // 2nd abstract
}
```

```
error: Unexpected @FunctionalInterface annotation
  Operation is not a functional interface
    multiple non-overriding abstract methods
    found in interface Operation
```

Custom Annotations and Reflection

```java
@Retention(RetentionPolicy.RUNTIME)  // read at run time
@Target(ElementType.METHOD)          // methods only
@interface Task {
    String author();
    int priority() default 1;
}
class Jobs {
    @Task(author = "Ayşe", priority = 3)
    public void backup() { System.out.println("Backup done"); }
    public void clean() { System.out.println("Cleaned"); }
}
// in main (reflection, java.lang.reflect.Method):
Jobs jobs = new Jobs();
for (Method m : Jobs.class.getDeclaredMethods()) {
    Task g = m.getAnnotation(Task.class);
    if (g != null) {
        System.out.println(m.getName() + ", "
            + g.author() + ", " + g.priority());
        m.invoke(jobs);
    }
}
```

Output

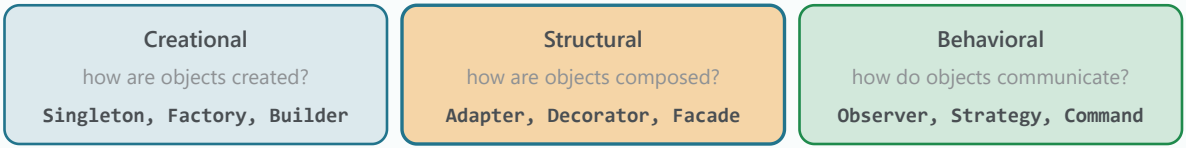
```
backup, Ayşe, 3
Backup done
```

`@Retention`	Where is it kept?
`SOURCE`	Source only (`@Override`)
`CLASS`	In the .class file; invisible to reflection
`RUNTIME`	Read via reflection at run time

- **Reflection**: a program inspects its own classes, methods and annotations at run time.
- It weakens compiler checks and slows code down; use it only when truly needed, as frameworks do.

Design Patterns

- A **design pattern** is a proven, reusable **solution template** for a recurring design problem. It is not code to copy but a shared **vocabulary** that names the problem and its trade-offs.
- Patterns separate responsibilities and **reduce coupling** between classes: one part can be **changed** without touching the others, and tested on its own.



A pattern helps only if it solves a **real problem**. An unnecessary pattern just complicates the program. Patterns are concrete applications of more general design **principles** such as “low coupling” and “composition over inheritance”.

Singleton: A Single Instance

```
public final class Settings {
  private static final Settings INSTANCE = new Settings();
  private Settings() {
    System.out.println("Settings created");
  }
  public static Settings getInstance() { return INSTANCE; }
}
// main:
System.out.println("main started");
Settings a = Settings.getInstance();
Settings b = Settings.getInstance();
System.out.println(a == b);
```

Output

```
main started
Settings created
true
```

- The class has **only one** instance, accessed from a single place.
- `private` constructor: `new Settings()` from outside does not compile.
- The JVM creates the instance once, when the class is first used: this is thread-safe.
- It acts like a hidden global variable: swapping in another object gets hard. Passing the object to the constructor (dependency injection) is usually better.

Factory

```
interface Shape { double area(); }
class Square implements Shape { … }   // area: a * a
class Circle implements Shape { … }  // area: π r²

class ShapeFactory {
  static Shape create(String kind, double size) {
    switch (kind.toLowerCase(Locale.ROOT)) {
      case "square":  return new Square(size);
      case "circle": return new Circle(size);
      default:       return null;  // unknown kind
    }
  }
}
Shape s = ShapeFactory.create("Square", 3);
System.out.println(s.area());   // 9.0
```

- Object creation is gathered in **one place**; the caller knows the **interface** (`Shape`), not the concrete class (`Square`).
- Adding a new shape changes only the factory. For an unknown kind it returns `null` ; the caller checks the result.
- `Locale.ROOT` : under a Turkish locale `"CIRCLE".toLowerCase()` gives `"cırcle"` , which does not match.
- Writing a factory for objects that are already simple to create is needless complexity.

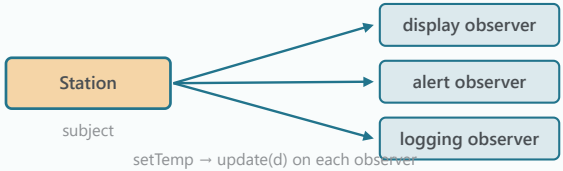
Unknown kind

```
Shape u = ShapeFactory.create("triangle",
2);
System.out.println(u == null);   // true
```

Observer

```
interface Observer { void update(int degrees); }

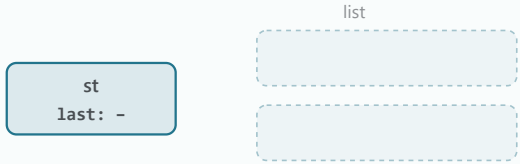
class Station {
  private final List<Observer> list = new ArrayList<>();
  void add(Observer g) { list.add(g); }
  void setTemp(int d) {
    for (Observer g : list) g.update(d);
  }
}
```



- A **one-to-many** relationship: when the subject changes, all registered observers are notified.
- The subject does not know the observers' concrete classes; it calls only the interface.
- Observers can be small objects that print to the console: one shows the reading, one raises an alert, one logs it.

The Observer Pattern Step by Step

```
1  interface Observer { void update(int degrees); }
2  class Station {
3    private final List<Observer> list = new ArrayList<>();
4    void add(Observer g) { list.add(g); }
5    void setTemp(int d) {
6      for (Observer g : list) g.update(d);
7    }
8  }
9  // main:
▶ Station st = new Station();
11 st.add(d -> System.out.println("Display: " + d + " °C"));
12 st.add(d -> { if (d > 30) System.out.println("Alert:
   hot!"); });
13 st.setTemp(25);
14 st.setTemp(34);
```



The subject object is created; the observer list is empty.

Step 1 / 9

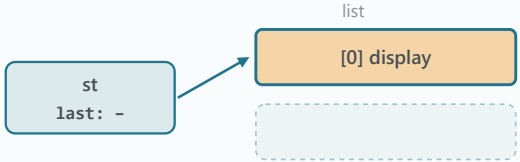
VARIABLES

`st = Station` `list = []`

OUTPUT

The Observer Pattern Step by Step

```
1  interface Observer { void update(int degrees); }
2  class Station {
3    private final List<Observer> list = new ArrayList<>();
▶    void add(Observer g) { list.add(g); }
5    void setTemp(int d) {
6      for (Observer g : list) g.update(d);
7    }
8  }
9  // main:
10 Station st = new Station();
▶  st.add(d -> System.out.println("Display: " + d + " °C"));
12 st.add(d -> { if (d > 30) System.out.println("Alert:
   hot!"); });
13 st.setTemp(25);
14 st.setTemp(34);
```



The first observer is a lambda implementing the single method of the `Observer` interface. It is added to the list; it does **not run**.

Step 2 / 9

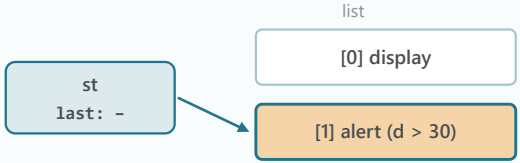
VARIABLES

`list = [display]`

OUTPUT

The Observer Pattern Step by Step

```
1  interface Observer { void update(int degrees); }
2  class Station {
3    private final List<Observer> list = new ArrayList<>();
▶    void add(Observer g) { list.add(g); }
5    void setTemp(int d) {
6      for (Observer g : list) g.update(d);
7    }
8  }
9  // main:
10 Station st = new Station();
11 st.add(d -> System.out.println("Display: " + d + " °C"));
▶ st.add(d -> { if (d > 30) System.out.println("Alert:
   hot!"); });
13 st.setTemp(25);
14 st.setTemp(34);
```



The second observer is added: it speaks up only above 30 degrees.

Step 3 / 9

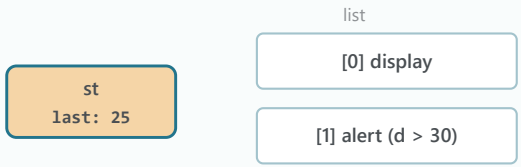
VARIABLES

`list = [display, alert]`

OUTPUT

The Observer Pattern Step by Step

```
1  interface Observer { void update(int degrees); }
2  class Station {
3    private final List<Observer> list = new ArrayList<>();
4    void add(Observer g) { list.add(g); }
5    void setTemp(int d) {
▶      for (Observer g : list) g.update(d);
7    }
8  }
9  // main:
10 Station st = new Station();
11 st.add(d -> System.out.println("Display: " + d + " °C"));
12 st.add(d -> { if (d > 30) System.out.println("Alert:
   hot!"); });
▶  st.setTemp(25);
14 st.setTemp(34);
```



`setTemp(25)` : the loop walks the list from the start.

Step 4 / 9

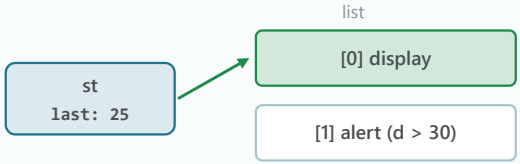
VARIABLES

d = **25**

OUTPUT

The Observer Pattern Step by Step

```
1  interface Observer { void update(int degrees); }
2  class Station {
3    private final List<Observer> list = new ArrayList<>();
4    void add(Observer g) { list.add(g); }
5    void setTemp(int d) {
▶      for (Observer g : list) g.update(d);
7    }
8  }
9  // main:
10 Station st = new Station();
▶ st.add(d -> System.out.println("Display: " + d + " °C"));
12 st.add(d -> { if (d > 30) System.out.println("Alert:
   hot!"); });
13 st.setTemp(25);
14 st.setTemp(34);
```



The display observer's `update(25)` is called.

Step 5 / 9

VARIABLES

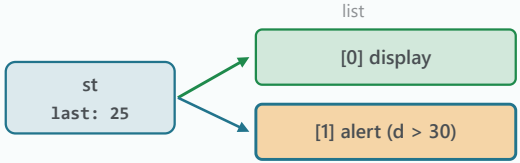
`d = 25` `g = display`

OUTPUT

```
Display: 25 °C
```

The Observer Pattern Step by Step

```
1  interface Observer { void update(int degrees); }
2  class Station {
3    private final List<Observer> list = new ArrayList<>();
4    void add(Observer g) { list.add(g); }
5    void setTemp(int d) {
▶      for (Observer g : list) g.update(d);
7    }
8  }
9  // main:
10 Station st = new Station();
11 st.add(d -> System.out.println("Display: " + d + " °C"));
▶ st.add(d -> { if (d > 30) System.out.println("Alert:
   hot!"); });
13 st.setTemp(25);
14 st.setTemp(34);
```



The alert observer is notified too, but 25 > 30 is false: it prints nothing.

Step 6 / 9

VARIABLES

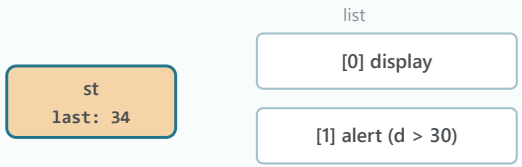
`d = 25` `g = alert`

OUTPUT

```
Display: 25 °C
```

The Observer Pattern Step by Step

```
1  interface Observer { void update(int degrees); }
2  class Station {
3    private final List<Observer> list = new ArrayList<>();
4    void add(Observer g) { list.add(g); }
5    void setTemp(int d) {
▶      for (Observer g : list) g.update(d);
7    }
8  }
9  // main:
10 Station st = new Station();
11 st.add(d -> System.out.println("Display: " + d + " °C"));
12 st.add(d -> { if (d > 30) System.out.println("Alert:
   hot!"); });
13 st.setTemp(25);
▶  st.setTemp(34);
```



A new reading: 34.

Step 7 / 9

VARIABLES

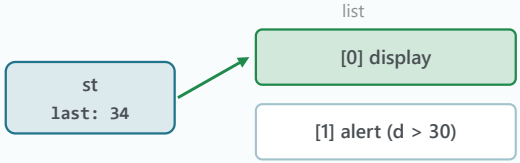
d = 34

OUTPUT

```
Display: 25 °C
```

The Observer Pattern Step by Step

```
1  interface Observer { void update(int degrees); }
2  class Station {
3    private final List<Observer> list = new ArrayList<>();
4    void add(Observer g) { list.add(g); }
5    void setTemp(int d) {
▶      for (Observer g : list) g.update(d);
7    }
8  }
9  // main:
10 Station st = new Station();
▶ st.add(d -> System.out.println("Display: " + d + " °C"));
12 st.add(d -> { if (d > 30) System.out.println("Alert:
   hot!"); });
13 st.setTemp(25);
14 st.setTemp(34);
```



The display prints the new value.

Step 8 / 9

VARIABLES

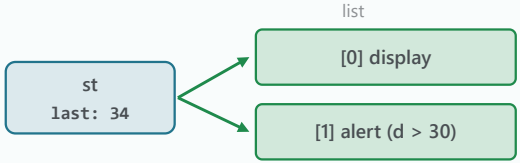
d = **34** g = **display**

OUTPUT

```
Display: 25 °C
Display: 34 °C
```

The Observer Pattern Step by Step

```
1  interface Observer { void update(int degrees); }
2  class Station {
3    private final List<Observer> list = new ArrayList<>();
4    void add(Observer g) { list.add(g); }
5    void setTemp(int d) {
▶      for (Observer g : list) g.update(d);
7    }
8  }
9  // main:
10 Station st = new Station();
11 st.add(d -> System.out.println("Display: " + d + " °C"));
▶ st.add(d -> { if (d > 30) System.out.println("Alert:
   hot!"); });
13 st.setTemp(25);
14 st.setTemp(34);
```



This time the condition holds and the alert is printed. `Station` never knew what any observer does; adding a new observer does not change its code.

Step 9 / 9

VARIABLES

`d = 34` `g = alert`

OUTPUT

```
Display: 25 °C
Display: 34 °C
Alert: hot!
```

Strategy

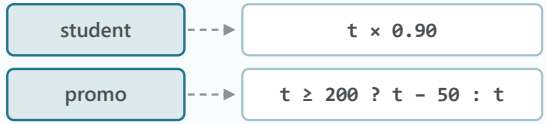
```
interface Discount { double apply(double amount); }

class StudentDiscount implements Discount {
  public double apply(double amount) { return amount * 0.90; }
}
class Checkout {
  private final Discount strategy;
  Checkout(Discount strategy) { this.strategy = strategy; }
  double due(double amount) {
    return strategy.apply(amount);
  }
}
Checkout k = new Checkout(new StudentDiscount());
System.out.println(k.due(250));   // 225.0
```

- Interchangeable algorithms are placed behind a common **interface**.
- Behavior changes with a different strategy object while `Checkout`'s code stays the same.
- Since the interface has one method, a strategy can also be a **lambda**: `new Checkout(t -> t * 0.90)`.
- Sorting with a `Comparator` is also a strategy.

The Strategy Pattern Step by Step

```
1  interface Discount { double apply(double amount); }
2  class Checkout {
3    private final Discount strategy;
4    Checkout(Discount strategy) { this.strategy = strategy;
   }
5    double due(double amount) { return
   strategy.apply(amount); }
6  }
7  // main:
▶  Checkout student = new Checkout(t -> t * 0.90);
9  Checkout promo = new Checkout(t -> t >= 200 ? t - 50 : t);
10 System.out.println(student.due(250));
11 System.out.println(promo.due(250));
12 System.out.println(promo.due(120));
13 // fixed strategy: Checkout tested alone
14 System.out.println(new Checkout(t -> 7).due(100));
```



`Checkout` does not compute the discount itself; it asks the **strategy** object passed to its constructor. The first checkout gets a 10% discount.

Step 1 / 6

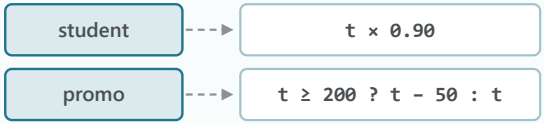
VARIABLES

`student = Checkout(10%)`

OUTPUT

The Strategy Pattern Step by Step

```
1  interface Discount { double apply(double amount); }
2  class Checkout {
3    private final Discount strategy;
4    Checkout(Discount strategy) { this.strategy = strategy;
   }
5    double due(double amount) { return
   strategy.apply(amount); }
6  }
7  // main:
8  Checkout student = new Checkout(t -> t * 0.90);
▶  Checkout promo = new Checkout(t -> t >= 200 ? t - 50 : t);
10 System.out.println(student.due(250));
11 System.out.println(promo.due(250));
12 System.out.println(promo.due(120));
13 // fixed strategy: Checkout tested alone
14 System.out.println(new Checkout(t -> 7).due(100));
```



The same `Checkout` class with another strategy: 50 TL off for 200 TL or more.

Step 2 / 6

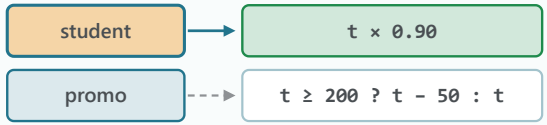
VARIABLES

`promo = Checkout(−50 from 200)`

OUTPUT

The Strategy Pattern Step by Step

```
1  interface Discount { double apply(double amount); }
2  class Checkout {
3    private final Discount strategy;
4    Checkout(Discount strategy) { this.strategy = strategy;
   }
▶    double due(double amount) { return
   strategy.apply(amount); }
6  }
7  // main:
8  Checkout student = new Checkout(t -> t * 0.90);
9  Checkout promo = new Checkout(t -> t >= 200 ? t - 50 : t);
▶  System.out.println(student.due(250));
11 System.out.println(promo.due(250));
12 System.out.println(promo.due(120));
13 // fixed strategy: Checkout tested alone
14 System.out.println(new Checkout(t -> 7).due(100));
```



due(250) → 225.0

`due` delegates to the strategy: 250 × 0.90 = 225.0.

Step 3 / 6

VARIABLES

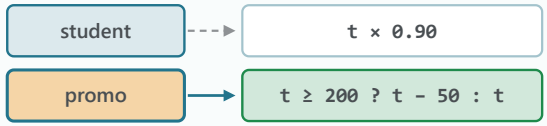
`amount = 250.0`

OUTPUT

```
225.0
```

The Strategy Pattern Step by Step

```
1  interface Discount { double apply(double amount); }
2  class Checkout {
3    private final Discount strategy;
4    Checkout(Discount strategy) { this.strategy = strategy;
   }
▶    double due(double amount) { return
   strategy.apply(amount); }
6  }
7  // main:
8  Checkout student = new Checkout(t -> t * 0.90);
9  Checkout promo = new Checkout(t -> t >= 200 ? t - 50 : t);
10 System.out.println(student.due(250));
▶  System.out.println(promo.due(250));
12 System.out.println(promo.due(120));
13 // fixed strategy: Checkout tested alone
14 System.out.println(new Checkout(t -> 7).due(100));
```



due(250) → 200.0

The promo strategy: 250 ≥ 200 → 250 − 50 = 200.0.

Step 4 / 6

VARIABLES

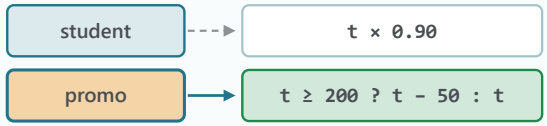
`amount = 250.0`

OUTPUT

```
225.0
200.0
```

The Strategy Pattern Step by Step

```
1  interface Discount { double apply(double amount); }
2  class Checkout {
3    private final Discount strategy;
4    Checkout(Discount strategy) { this.strategy = strategy;
   }
▶    double due(double amount) { return
   strategy.apply(amount); }
6  }
7  // main:
8  Checkout student = new Checkout(t -> t * 0.90);
9  Checkout promo = new Checkout(t -> t >= 200 ? t - 50 : t);
10 System.out.println(student.due(250));
11 System.out.println(promo.due(250));
▶  System.out.println(promo.due(120));
13 // fixed strategy: Checkout tested alone
14 System.out.println(new Checkout(t -> 7).due(100));
```



due(120) → 120.0

120 < 200: no discount. The behavior changed without touching `Checkout`'s code.

Step 5 / 6

VARIABLES

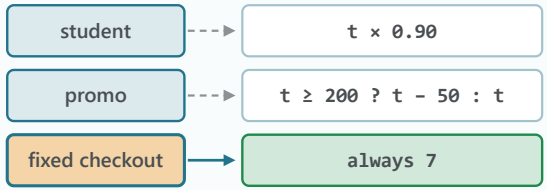
`amount = 120.0`

OUTPUT

```
225.0
200.0
120.0
```

The Strategy Pattern Step by Step

```
1  interface Discount { double apply(double amount); }
2  class Checkout {
3    private final Discount strategy;
4    Checkout(Discount strategy) { this.strategy = strategy;
   }
5    double due(double amount) { return
   strategy.apply(amount); }
6  }
7  // main:
8  Checkout student = new Checkout(t -> t * 0.90);
9  Checkout promo = new Checkout(t -> t >= 200 ? t - 50 : t);
10 System.out.println(student.due(250));
11 System.out.println(promo.due(250));
12 System.out.println(promo.due(120));
13 // fixed strategy: Checkout tested alone
▶  System.out.println(new Checkout(t -> 7).due(100));
```



due(100) → 7.0

A **fixed** strategy is plugged in instead of a real discount rule: whatever the amount, the result is 7.0. This shows only that `Checkout` delegates correctly to its strategy.

Step 6 / 6

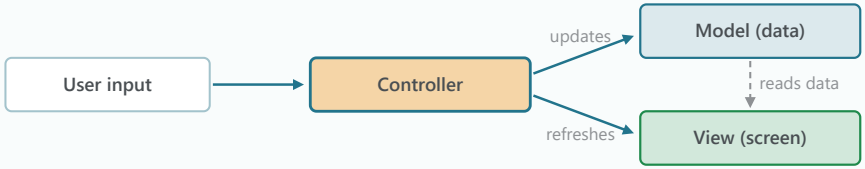
VARIABLES

`amount = 100.0`

OUTPUT

```
225.0
200.0
120.0
7.0
```

MVC: Model, View, Controller



- **Model**: data and rules (`Student` , `getName()`); unaware of the screen, testable without it.
- **View**: only displays (`StudentView.show(name)` prints to the console).
- **Controller**: takes input and manages the model and view; `update()` reads the model and passes the result to the view.
- Frameworks implement MVC in different ways; there is no single “right” class structure.

```
Student m = new Student("Ayşe");
StudentView g = new StudentView();
StudentController d =
    new StudentController(m, g);
d.update();  // g.show(m.getName())
```

Output

```
Student: Ayşe
```

Patterns: When Not to Use Them

Turning a Singleton into a global variable

Mutable state reachable from everywhere couples the parts of a program to each other.

Needless factory

Writing a separate class for something as simple as `new Point(3, 4)`.

Needless observer

Building an event system where one direct method call would do; the flow becomes hard to follow.

Using a pattern for its own sake

Problem first, pattern second: “Which change does it make easier? Which dependency does it reduce?”

Strategies and dependencies passed in from outside let you replace a real part with a simple **fake object** (like plugging a fixed strategy into `Checkout`). A class that is hard to test on its own usually needs its design revisited.

Summary

Concept	Essence
Abstraction	Hide the details and define “what”; leave “how” to subclasses
Abstract class	`abstract class`; no instances; may hold fields, constructors, abstract and concrete methods
Abstract method	No body, ends with `;`; a concrete subclass must implement it
Interface	A contract; methods implicitly `public abstract`, fields `public static final`
implements	One `extends`, many `implements`; implementing methods are `public`
Lambda	`(parameters) -> expression`; assigned to a functional interface (Java 8+)
default / static / private	Interface methods with bodies (Java 8 / 8 / 9)
sealed	Permitted subtypes via `permits`; each is `final`, `sealed` or `non-sealed`
Generics	`Box<T>`, `<T extends B>`, wildcard `?`; checked at compile time, type information erased afterwards
Annotations	Metadata such as `@Override`; `RUNTIME` annotations are read via reflection
Design patterns	Singleton, Factory, Observer, Strategy, MVC: proven solution templates

SUMMARY

Related Pages

Polymorphism

The basics of dynamic binding with references of interface and abstract class types.

Writing `@Override` when implementing an abstract or interface method catches a misspelled name or wrong parameters at once, as a compile error.

SUMMARY

Common Mistakes (1): Abstract Classes

Instantiating an abstract class

`new Shape("?")` → `Shape is abstract; cannot be instantiated`. Use a concrete subclass.

Forgetting an abstract method

`Circle is not abstract and does not override abstract method draw() in Shape`. Write the method or make the class `abstract`.

A body on an abstract method

`abstract void draw() { }` → `abstract methods cannot have a body`. Either `abstract` or a body.

Not making the class abstract

`class Shape { abstract void draw(); }` → `Shape is not abstract and does not override abstract method draw() in Shape`.

Conflicting modifiers

`abstract final`, `private abstract`, `static abstract` → `illegal combination of modifiers`.

Getting the signature wrong

`double Area()` or `double area(int x)` is a new method; the abstract `area()` is still missing. `@Override` catches this.

SUMMARY

Common Mistakes (2): Interfaces

Forgetting `public`

`void draw() { … }` → `attempting to assign weaker access privileges; was public`. Write interface methods as `public`.

Changing a constant

`GeometricShape.SIDES = 5;` → `cannot assign a value to static final variable SIDES`.

Extending two classes

`class C extends A, B` → `'{' expected`. One `extends`, many `implements`.

A method the reference type lacks

`Flyable u = new Duck(); u.swim();` → `cannot find symbol`. Call a method through a reference whose type knows it.

Conflicting default methods

`class Clerk inherits unrelated defaults for greet()`. Override the method in the class.

Sealing errors

Unpermitted subtype: `class is not allowed to extend sealed class: Animal`. Permitted subtype without a modifier: `sealed, non-sealed or final modifiers expected`.

SUMMARY

Practice Problems (1): Does It Compile?

```
// (a)
abstract class A {
    void f() { System.out.println("f"); }
}
// (b)
class B { abstract void g(); }
// (c)
interface I { int X = 1; }
...  I.X++;
// (d)
interface J { void h(); }
class C implements J { void h() { } }
// (e)
abstract class D { abstract void k(); }
abstract class E extends D { }
```

1. For each fragment (a)–(e): does it compile? If not, what is the error?
2. Which of the compiling fragments can be instantiated with `new`?

▶ **Show answers**

Practice Problems (2): What Is the Output?

```
interface Noisy {
    default String sound() { return "..."; }
}
abstract class Animal implements Noisy {
    Animal() { System.out.print("A "); }
    abstract String name();
}
class Cat extends Animal {
    Cat() { System.out.print("C "); }
    String name() { return "cat"; }
    public String sound() { return "meow"; }
}
class Fish extends Animal {
    String name() { return "fish"; }
}
// in main:
Animal[] d = { new Cat(), new Fish() };
System.out.println();
for (Animal h : d)
    System.out.println(h.name() + ": " + h.sound());
```

1. What does the program print?
2. Why does `Fish` compile even though it does not write `sound()`?
3. What happens if `public` is removed from `sound()` in `Cat`?
4. Does `Noisy s = new Fish(); s.name();` compile?

▶ **Show answers**

SUMMARY

Exercises (1)

1. Write an abstract class `Employee` : a `name` field, a constructor, an abstract `double salary()` and an `info()` that prints the name and salary. Subclasses: `FullTime` (30000 a month), `Hourly` (120 hours × 200). Call `info()` on an `Employee[]` array.
2. Interface `Shape2D` : an abstract `double area()` and a default `summary()` returning `"Area: " + area()` . What does `summary()` return for `Triangle(6, 4)` ?
3. `interface Calc { static int sum(int a, int b) … }` and `class Simple implements Calc {}` : what do `Calc.sum(3, 4)` and `Simple.sum(3, 4)` give?

▶ **Answers / hints**

Exercises (2)

4. Write `sealed abstract class Shape permits Square, Circle`, with Square `final` and Circle `non-sealed`. Do `Ellipse extends Circle` and `Triangle extends Shape` compile?
5. Abstract class or interface? (a) a `Vehicle` with a plate number and a constructor; (b) a shared `save()` capability of `Student` and `Invoice`; (c) a `Fish` that is an Animal and can swim.
6. Inside `class Square implements GeometricShape` someone wrote `public void Draw() { … }`. What happens? And with `@Override` above the method?

▶ **Answers / hints**

References

- Oracle, *The Java Tutorials*: Abstract Methods and Classes; Interfaces; Default Methods; Lambda Expressions.
- J. Gosling et al., *The Java Language Specification*, Java SE 21 Edition (8.1.1.1 abstract Classes, 8.1.1.2 sealed Classes, Chapter 9: Interfaces, 15.27 Lambda Expressions).
- Oracle, *JEP 409: Sealed Classes* (Java 17) and *JEP 441: Pattern Matching for switch* (Java 21).
- P. Deitel and H. Deitel, *Java: How to Program*, 11th ed., Pearson (Chapter 10: Object-Oriented Programming: Polymorphism and Interfaces).
- K. Sierra and B. Bates, *Head First Java*, 3rd ed., O'Reilly (Chapter 8: Interfaces and Abstract Classes).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12th ed., Pearson (Chapters 5–6).
- E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

The End

Chapter 4.5: Abstract Classes and Interfaces

Previous: Chapter 4.4: Polymorphism · Next: Chapter 5: Collections