PROGRAMMING IN JAVA

Lecture 9

Interfaces and Multiple Inheritance in Java

Interface Contracts, `implements`, Multiple Interface Implementation, Default Methods, Functional Interfaces, and Ordering

Lecturer: Arshyn Altybay

Department of Computer Engineering,
Almaty Technological University, Almaty, Kazakhstan

1. Learning Objectives

After completing this lecture, students should be able to:

1. explain the purpose of an interface in Java;
2. distinguish between classes, abstract classes, and interfaces;
3. declare and implement interfaces;
4. use the `implements` keyword correctly;
5. create classes that implement more than one interface;
6. explain how Java supports multiple-type inheritance through interfaces;
7. use interface references for polymorphic programming;
8. create interface inheritance hierarchies;
9. define and use default methods;
10. define static and private methods in interfaces;
11. resolve default-method conflicts;
12. explain the purpose of functional interfaces;
13. use lambda expressions with simple functional interfaces;
14. implement `Comparable` for natural ordering;
15. use `Comparator` for alternative orderings;
16. select appropriately between an abstract class and an interface.

2. Introduction

In the previous lecture, we studied inheritance and polymorphism using classes.

A class can extend only one direct superclass:

```
class Student extends Person {
}
```

Java does **not** support multiple inheritance of classes.

The following is illegal:

```
class C extends A, B {   // invalid Java
}
```

However, real software often requires one class to satisfy several independent behavioral contracts.

For example, one object may be:

- printable;
- sortable;
- storable;
- comparable;
- payable.

Java solves this problem using **interfaces**.

3. What Is an Interface?

An interface is a Java type that defines a contract.

It describes **what an object can do** without necessarily specifying **how the object does it**.

Example:

```
interface Printable {

    void print();
}
```

A class that implements this interface promises to provide an implementation of `print()`.

```
class Report implements Printable {

    @Override
    public void print() {
        System.out.println("Printing report");
    }
}
```

The interface describes the required behavior.

The class provides the actual implementation.

4. Declaring an Interface

The general syntax is:

```
interface InterfaceName {

    // constants
    // abstract methods
    // default methods
    // static methods
    // private helper methods
}
```

A simple example:

```
interface Payable {

    double calculatePayment();
}
```

5. The `implements` Keyword

A class implements an interface using the keyword `implements`.

```
class Employee implements Payable {

    @Override
    public double calculatePayment() {
        return 300000.0;
    }
}
```

Important rule:

An implementing method must be public.

Interface abstract methods are public by contract.

Therefore this is incorrect:

```
class Employee implements Payable {

    // Incorrect: weaker access
    double calculatePayment() {
        return 300000.0;
    }
}
```

Example 1. Basic Interface

```
public class Main {

    interface Printable {

        void print();
    }

    static class Report
            implements Printable {

        @Override
        public void print() {
            System.out.println(
                    "Printing report"
            );
        }
    }

    public static void main(String[] args) {

        Report report =
                new Report();

        report.print();
    }
}
```

Expected output

```
Printing report
```

6. Interface References

Like a superclass type, an interface can be used as a reference type.

```
Printable item = new Report();
```

The variable type is the interface.

The actual object is an instance of a class that implements the interface.

This is an important form of polymorphism.

Example 2. Polymorphism Through an Interface

```
1  public class Main {
2
3      interface Printable {
4
5          void print();
6      }
7
8      static class Report
9              implements Printable {
10
11         @Override
12         public void print() {
13             System.out.println(
14                     "Report printed"
15             );
16         }
17     }
18
19     static class Invoice
20             implements Printable {
21
22         @Override
23         public void print() {
24             System.out.println(
25                     "Invoice printed"
26             );
27         }
28     }
29
30     public static void main(String[] args) {
31
32         Printable p1 =
33                 new Report();
34
35         Printable p2 =
36                 new Invoice();
37
38         p1.print();
39         p2.print();
40     }
41 }
```

Expected output

```
Report printed
Invoice printed
```

The client code can work with objects through the common interface contract.

7. Multiple Interface Implementation

A class may implement more than one interface.

Example:

```
interface Printable {
    void print();
}

interface Scannable {
    void scan();
}

class MultiFunctionDevice
        implements Printable, Scannable {

    @Override
    public void print() {
        System.out.println("Printing");
    }

    @Override
    public void scan() {
        System.out.println("Scanning");
    }
}
```

This is how Java supports a form of multiple inheritance.

8. Multiple Inheritance in Java

Java does not support:

multiple inheritance of classes

but Java does support:

multiple implementation of interfaces

For example:

```
class Device
        implements Printable,
                   Scannable,
                   Connectable {
}
```

A class may therefore participate in several independent behavioral contracts.

Example 3. Multiple Interfaces

```
1  public class Main {
2
3      interface Printable {
4
5          void print();
6      }
7
8      interface Scannable {
9
10         void scan();
11     }
12
13     static class PrinterScanner
14             implements Printable,
15                        Scannable {
16
17         @Override
18         public void print() {
19             System.out.println(
20                     "Document printed"
21             );
22         }
```

Example 3 (continued)

```
23
24          @Override
25          public void scan() {
26              System.out.println(
27                      "Document scanned"
28              );
29          }
30      }
31
32      public static void main(String[] args) {
33
34          PrinterScanner device =
35                  new PrinterScanner();
36
37          device.print();
38          device.scan();
39      }
40  }
```

Expected output

```
Document printed
Document scanned
```

9. Interface Inheritance

An interface can extend another interface.

Example:

```
interface Readable {

    void read();
}

interface Editable extends Readable {

    void edit();
}
```

A class implementing `Editable` must implement both methods.

```
class Document implements Editable {

    @Override
    public void read() {
        System.out.println("Reading");
    }

    @Override
    public void edit() {
        System.out.println("Editing");
    }
}
```

10. Multiple Interface Inheritance

Unlike classes, an interface may extend more than one interface.

```
interface Printable {
    void print();
}

interface Scannable {
    void scan();
}

interface OfficeDevice
        extends Printable,
                Scannable {
}
```

A class implementing `OfficeDevice` must satisfy both contracts.

Example 4. Interface Inheritance

```
1 public class Main {
2
3     interface Printable {
4
5         void print();
6     }
7
8     interface Scannable {
```

Example 4 (continued)

```
9
10          void scan();
11      }
12
13      interface OfficeDevice
14              extends Printable,
15                      Scannable {
16
17          void status();
18      }
19
20      static class Machine
21              implements OfficeDevice {
22
23          @Override
24          public void print() {
25              System.out.println("Print");
26          }
27
28          @Override
29          public void scan() {
30              System.out.println("Scan");
31          }
32
33          @Override
34          public void status() {
35              System.out.println("Ready");
36          }
37      }
38
39      public static void main(String[] args) {
40
41          OfficeDevice device =
42                  new Machine();
43
44          device.print();
45          device.scan();
46          device.status();
47      }
48  }
```

Expected output

```
Print
Scan
```

```
Ready
```

11. Fields in Interfaces

Fields declared in an interface are implicitly:

- `public`;
- `static`;
- `final`.

Therefore:

```
interface Configuration {

    int MAX_USERS = 100;
}
```

is conceptually equivalent to:

```
interface Configuration {

    public static final
    int MAX_USERS = 100;
}
```

The value is a constant.

It cannot be changed.

Example 5. Interface Constant

```
1  public class Main {
2
3      interface Limits {
4
5          int MAX_STUDENTS = 30;
6      }
7
8      public static void main(String[] args) {
9
10         System.out.println(
11                 Limits.MAX_STUDENTS
12         );
13     }
14 }
```

Expected output

```
30
```

12. Default Methods

Originally, Java interfaces contained only abstract methods and constants.

Modern Java allows an interface to define a method with an implementation using the keyword `default`.

Example:

```
interface Printable {

    void print();

    default void preview() {
        System.out.println("Preview");
    }
}
```

A class implementing the interface automatically receives the default method.

It may also override it.

13. Why Default Methods Exist

Default methods allow an interface to evolve without forcing every existing implementing class to implement a newly added method.

They are especially useful in large APIs.

However, interfaces should still mainly describe capabilities and contracts rather than become general-purpose implementation containers.

Example 6. Default Method

```
1  public class Main {
2
3      interface Printable {
4
5          void print();
6
7          default void preview() {
8              System.out.println(
9                      "Default preview"
10             );
11         }
12     }
13
14     static class Report
15             implements Printable {
16
17         @Override
18         public void print() {
19             System.out.println(
20                     "Printing report"
21             );
22         }
23     }
24
25     public static void main(String[] args) {
26
27         Report report =
28                 new Report();
29
30         report.preview();
31         report.print();
32     }
33 }
```

Expected output

```
Default preview
Printing report
```

14. Overriding a Default Method

An implementing class may replace a default implementation.

```
interface Printable {

    default void preview() {
        System.out.println(
                "Interface preview"
        );
    }
}

class Report implements Printable {

    @Override
    public void preview() {
        System.out.println(
                "Report preview"
        );
    }
}
```

The class implementation takes precedence.

15. Default Method Conflicts

Suppose two interfaces define default methods with the same signature.

```
interface A {

    default void show() {
        System.out.println("A");
    }
}

interface B {

    default void show() {
        System.out.println("B");
    }
}
```

A class implementing both must resolve the conflict explicitly.

Example 7. Resolving a Default-Method Conflict

```
1  public class Main {
2
3      interface A {
4
5          default void show() {
6              System.out.println("A");
7          }
8      }
9
10     interface B {
11
12         default void show() {
13             System.out.println("B");
14         }
15     }
16
17     static class C
18             implements A, B {
19
20         @Override
21         public void show() {
22
23             A.super.show();
24             B.super.show();
```

Example 7 (continued)

```
25
26              System.out.println("C");
27          }
28      }
29
30      public static void main(String[] args) {
31
32          C object = new C();
33
34          object.show();
35      }
36  }
```

Expected output

```
A
B
C
```

The syntax:

```
A.super.show();
```

calls the default method from a specific interface.

16. Static Methods in Interfaces

An interface may contain static methods.

```
interface NumberUtil {

    static boolean isEven(int n) {
        return n % 2 == 0;
    }
}
```

The method is called through the interface name:

```
NumberUtil.isEven(10);
```

Static interface methods are not inherited as instance methods by implementing classes.

Example 8. Static Interface Method

```
1  public class Main {
2
3      interface MathCheck {
4
5          static boolean positive(
6                  int value) {
7
8              return value > 0;
9          }
10     }
11
12     public static void main(String[] args) {
13
14         System.out.println(
15                 MathCheck.positive(15)
16         );
17     }
18 }
```

Expected output

```
true
```

17. Private Methods in Interfaces

Modern Java also allows private methods inside interfaces.

They are mainly used as helper methods shared by default or static methods.

Example:

```
interface Logger {

    default void info(String message) {
        write("INFO", message);
    }

    default void error(String message) {
        write("ERROR", message);
    }

    private void write(
            String level,
            String message) {

        System.out.println(
                level + ": " + message
        );
    }
}
```

The private method is an implementation detail of the interface.

Implementing classes cannot call it directly.

Example 9. Private Interface Helper

```
1  public class Main {
2
3      interface Logger {
4
5          default void info(
6                  String message) {
7
8              log("INFO", message);
9          }
10
11         default void warning(
12                 String message) {
```

Example 9 (continued)

```
13
14              log("WARNING", message);
15          }
16
17          private void log(
18                  String level,
19                  String message) {
20
21              System.out.println(
22                      level + ": " +
23                      message
24              );
25          }
26      }
27
28      static class Application
29              implements Logger {
30      }
31
32      public static void main(String[] args) {
33
34          Application app =
35                  new Application();
36
37          app.info("Started");
38          app.warning("Low memory");
39      }
40  }
```

Expected output

```
INFO: Started
WARNING: Low memory
```

18. Interfaces and Abstract Classes

Interfaces and abstract classes are related, but they serve different design purposes.

Feature	Abstract Class	Interface
Can contain instance fields	Yes	No ordinary instance fields
Can have constructors	Yes	No
Can have abstract methods	Yes	Yes
Can have concrete methods	Yes	Yes, via default/static/private methods
Class can use several?	Only one superclass	Many interfaces
Best suited for	shared base state/implementation	capability/contract
Inheritance keyword	`extends`	`implements`

19. When to Use an Abstract Class

An abstract class is often appropriate when:

- subclasses share common state;
- subclasses require common constructors;
- there is substantial shared implementation;
- the types form a strong is-a hierarchy.

Example:

Illustration

```
Employee
  |
  +-- FullTimeEmployee
  +-- PartTimeEmployee
```

20. When to Use an Interface

An interface is often appropriate when:

- unrelated classes share a capability;
- multiple behavioral contracts are needed;
- client code should depend on an abstraction;
- implementation may vary independently.

Examples:

Illustration

```
Printable
Comparable
Runnable
AutoCloseable
```

A class may extend one class and implement several interfaces at the same time.

Example 10. One Class Extending a Class and Implementing Interfaces

```
1  public class Main {
2
3      static class Person {
4
5          private String name;
6
7          public Person(String name) {
8              this.name = name;
9          }
10
11         public String getName() {
12             return name;
13         }
14     }
15
16     interface Printable {
17
18         void print();
19     }
20
21     interface Identifiable {
22
23         int getId();
24     }
```

Example 10 (continued)

```
25
26      static class Student
27              extends Person
28              implements Printable,
29                         Identifiable {
30
31          private int id;
32
33          public Student(
34                  int id,
35                  String name) {
36
37              super(name);
38              this.id = id;
39          }
40
41          @Override
42          public int getId() {
43              return id;
44          }
45
46          @Override
47          public void print() {
48
49              System.out.println(
50                      getId() + ": " +
51                      getName()
52              );
53          }
54      }
55
56      public static void main(String[] args) {
57
58          Student student =
59                  new Student(
60                          1001,
61                          "Aruzhan"
62                  );
63
64          student.print();
65      }
66  }
```

Expected output

```
1001: Aruzhan
```

21. Interface-Based Design

One of the most important uses of interfaces is to reduce coupling.

Consider a notification system.

Poor design:

```
class NotificationService {

    private EmailSender sender =
            new EmailSender();
}
```

The service is tightly coupled to one implementation.

A more flexible design uses an interface.

```
interface MessageSender {

    void send(String message);
}
```

Then several implementations are possible:

```
class EmailSender
        implements MessageSender {
}

class SmsSender
        implements MessageSender {
}
```

The service depends on the abstraction rather than one concrete class.

Example 11. Interface-Based Notification System

```
public class Main {

    interface MessageSender {

        void send(String message);
    }

    static class EmailSender
            implements MessageSender {

        @Override
        public void send(
                String message) {

            System.out.println(
                    "Email: " + message
            );
        }
    }

    static class SmsSender
            implements MessageSender {

        @Override
        public void send(
                String message) {

            System.out.println(
                    "SMS: " + message
            );
        }
    }

    static class NotificationService {

        private MessageSender sender;

        public NotificationService(
                MessageSender sender) {

            this.sender = sender;
        }

        public void notifyUser(
```

Example 11 (continued)

```
45                  String message) {
46
47              sender.send(message);
48          }
49      }
50
51      public static void main(String[] args) {
52
53          NotificationService s1 =
54                  new NotificationService(
55                          new EmailSender()
56                  );
57
58          NotificationService s2 =
59                  new NotificationService(
60                          new SmsSender()
61                  );
62
63          s1.notifyUser(
64                  "Exam tomorrow"
65          );
66
67          s2.notifyUser(
68                  "Class starts at 10"
69          );
70      }
71  }
```

Expected output

```
Email: Exam tomorrow
SMS: Class starts at 10
```

This design can be extended with new sender implementations without changing `NotificationService`.

22. Functional Interfaces

A **functional interface** contains exactly one abstract method.

Example:

```
@FunctionalInterface
interface Calculator {

    int calculate(int a, int b);
}
```

Functional interfaces are important because they can be implemented using lambda expressions.

23. The `@FunctionalInterface` Annotation

The annotation is optional but recommended.

```
@FunctionalInterface
interface Operation {

    int apply(int a, int b);
}
```

The compiler checks that the interface has exactly one abstract method.

Default and static methods do not count as abstract methods.

Example 12. Functional Interface and Lambda

```
1  public class Main {
2
3      @FunctionalInterface
4      interface Operation {
5
6          int apply(int a, int b);
7      }
8
9      public static void main(String[] args) {
10
11         Operation add =
12                 (a, b) -> a + b;
13
14         Operation multiply =
15                 (a, b) -> a * b;
16
17         System.out.println(
18                 add.apply(5, 3)
19         );
20
21         System.out.println(
22                 multiply.apply(5, 3)
23         );
24     }
25 }
```

Expected output

```
8
15
```

Lambda expressions will be studied in more detail in a later lecture.

24. Comparable

The standard Java interface `Comparable<T>` defines a natural ordering for objects.

Its main method is:

```
int compareTo(T other);
```

Typical interpretation:

- negative value: this object comes before the other;
- zero: objects are equal in ordering;
- positive value: this object comes after the other.

Example 13. Implementing `Comparable`

```
1  import java.util.Arrays;
2
3  public class Main {
4
5      static class Student
6              implements
7              Comparable<Student> {
8
9          private String name;
10         private double gpa;
11
12         public Student(
13                 String name,
14                 double gpa) {
15
16             this.name = name;
17             this.gpa = gpa;
18         }
19
20         @Override
21         public int compareTo(
22                 Student other) {
23
24             return name.compareTo(
25                     other.name
26             );
```

Example 13 (continued)

```
27          }
28
29          @Override
30          public String toString() {
31              return name +
32                      " (" + gpa + ")";
33          }
34      }
35
36      public static void main(String[] args) {
37
38          Student[] students = {
39              new Student("Dias", 3.5),
40              new Student("Aruzhan", 3.8),
41              new Student("Ali", 3.6)
42          };
43
44          Arrays.sort(students);
45
46          for (Student s : students) {
47              System.out.println(s);
48          }
49      }
50  }
```

Expected output

```
Ali (3.6)
Aruzhan (3.8)
Dias (3.5)
```

The natural ordering in this example is alphabetical by name.

25. Comparator

Sometimes one natural ordering is not enough.

For example, students may need to be sorted by:

- name;
- GPA;
- age;
- ID.

The interface `Comparator<T>` represents an external comparison strategy.

Its main method is:

```
int compare(T a, T b);
```

Example 14. Sorting with `Comparator`

```
import java.util.Arrays;
import java.util.Comparator;

public class Main {

    static class Student {

        private String name;
        private double gpa;

        public Student(
                String name,
                double gpa) {

            this.name = name;
            this.gpa = gpa;
        }

        public double getGpa() {
            return gpa;
        }

        @Override
        public String toString() {
            return name +
                    " (" + gpa + ")";
```

Example 14 (continued)

```
        }
    }

    public static void main(String[] args) {

        Student[] students = {
            new Student("Dias", 3.5),
            new Student("Aruzhan", 3.8),
            new Student("Ali", 3.6)
        };

        Comparator<Student> byGpa =
                (s1, s2) ->
                    Double.compare(
                        s2.getGpa(),
                        s1.getGpa()
                    );

        Arrays.sort(
                students,
                byGpa
        );

        for (Student s : students) {
            System.out.println(s);
        }
    }
}
```

Expected output

```
Aruzhan (3.8)
Ali (3.6)
Dias (3.5)
```

26. Comparable vs Comparator

Property	Comparable	Comparator
Package	`java.lang`	`java.util`
Main method	`compareTo()`	`compare()`
Defined where?	inside compared class	separate strategy/object
Number of orderings	usually one natural ordering	many alternative orderings
Typical use	default order	custom sorting

27. Marker Interfaces

Some interfaces historically contain no methods.

Such interfaces are called **marker interfaces**.

They indicate that a class has a particular property.

A traditional Java example is:

```
java.io.Serializable
```

It marks objects as eligible for Java object serialization.

Marker interfaces are less common in new designs because annotations can also express metadata, but they remain part of Java.

28. Interfaces in the Java Standard Library

Important standard interfaces include:

- `Comparable<T>`
- `Comparator<T>`
- `Runnable`
- `AutoCloseable`
- `Iterable<T>`
- `List<E>`
- `Set<E>`
- `Map<K,V>`
- `Collection<E>`

Many major Java APIs are designed around interfaces.

This allows multiple implementations to be used through a common contract.

29. Interface Segregation

A good interface should normally represent one focused capability.

Instead of creating one very large interface:

```
interface Machine {

    void print();
    void scan();
    void fax();
    void copy();
    void email();
    void connectWifi();
}
```

it may be better to create smaller interfaces:

```
interface Printable {
    void print();
}

interface Scannable {
```

```
    void scan();
}

interface Faxable {
    void fax();
}
```

Then classes implement only the capabilities they actually support.

This idea is related to the Interface Segregation Principle.

Example 15. Payment Strategy

```
public class Main {

    interface PaymentMethod {

        void pay(double amount);
    }

    static class CardPayment
            implements PaymentMethod {

        @Override
        public void pay(double amount) {
            System.out.println(
                "Card payment: " +
                amount
            );
        }
    }

    static class BankTransfer
            implements PaymentMethod {

        @Override
        public void pay(double amount) {
            System.out.println(
                "Bank transfer: " +
                amount
            );
        }
    }

    static class Checkout {

```

Example 15 (continued)

```
34          private PaymentMethod
35                  paymentMethod;
36
37          public Checkout(
38                  PaymentMethod
39                  paymentMethod) {
40
41              this.paymentMethod =
42                      paymentMethod;
43          }
44
45          public void complete(
46                  double total) {
47
48              paymentMethod.pay(total);
49          }
50      }
51
52      public static void main(String[] args) {
53
54          Checkout checkout =
55                  new Checkout(
56                      new CardPayment()
57                  );
58
59          checkout.complete(25000);
60      }
61  }
```

Expected output

```
Card payment: 25000.0
```

The checkout process depends on the `PaymentMethod` abstraction.

A new payment type can be added without changing `Checkout`.

Example 16. Interface-Based Repository

```
1  public class Main {
2
3      interface StudentRepository {
4
5          void save(String student);
6
7          String findById(int id);
```

Example 16 (continued)

```
8       }
9
10      static class MemoryRepository
11              implements StudentRepository {
12
13          private String stored;
14
15          @Override
16          public void save(
17                  String student) {
18
19              stored = student;
20          }
21
22          @Override
23          public String findById(
24                  int id) {
25
26              return stored;
27          }
28      }
29
30      static class StudentService {
31
32          private StudentRepository
33                  repository;
34
35          public StudentService(
36                  StudentRepository
37                  repository) {
38
39              this.repository =
40                      repository;
41          }
42
43          public void register(
44                  String student) {
45
46              repository.save(student);
47          }
48
49          public String find(int id) {
50              return repository.findById(id);
51          }
52      }
```

Example 16 (continued)

```
53
54      public static void main(String[] args) {
55
56          StudentRepository repository =
57                  new MemoryRepository();
58
59          StudentService service =
60                  new StudentService(
61                          repository
62                  );
63
64          service.register("Aruzhan");
65
66          System.out.println(
67                  service.find(1)
68          );
69      }
70  }
```

Expected output

```
Aruzhan
```

Later, the memory repository could be replaced by a file or database implementation without changing the service contract.

30. Common Mistakes

30.1 Mistake 1: Forgetting `public`

Incorrect:

```
interface Printable {
    void print();
}

class Report implements Printable {

    // Too restrictive
    void print() {
    }
}
```

Correct:

```
@Override
public void print() {
}
```

30.2 Mistake 2: Trying to Instantiate an Interface

Incorrect:

```
// Printable p = new Printable();
```

Interfaces are implemented by classes.

30.3 Mistake 3: Treating Interface Fields as Mutable State

Interface fields are constants.

They are not per-object mutable fields.

30.4 Mistake 4: Creating Very Large Interfaces

Large interfaces force classes to implement behavior they may not need.

Prefer smaller capability-focused interfaces.

30.5 Mistake 5: Using an Interface When Shared State Is Essential

If several classes strongly share common state and implementation, an abstract class may be more appropriate.

30.6 Mistake 6: Assuming Default Methods Remove the Need for Good Design

Default methods are useful, but they should not turn an interface into a large implementation-heavy class substitute.

31. Design Guidelines

When designing with interfaces:

1. define clear behavioral contracts;
2. keep interfaces focused;
3. depend on interfaces rather than concrete implementations when practical;
4. use multiple interfaces to represent independent capabilities;
5. use abstract classes when shared state and constructor logic are important;
6. use default methods carefully;
7. resolve default-method conflicts explicitly;
8. use functional interfaces for single-operation behavior;
9. define one natural ordering with `Comparable`;
10. use `Comparator` for alternative orderings;
11. avoid unnecessary coupling to implementation classes.

32. Summary

In this lecture, we learned that:

- an interface defines a behavioral contract;
- classes implement interfaces using `implements`;
- a class may implement multiple interfaces;
- Java does not support multiple inheritance of classes;
- Java supports multiple-type inheritance through interfaces;
- interfaces can be used as polymorphic reference types;

- interfaces may extend one or more other interfaces;
- interface fields are public static final constants;
- default methods provide reusable interface behavior;
- static interface methods belong to the interface itself;
- private interface methods can support internal reuse;
- conflicts between default methods must be resolved explicitly;
- functional interfaces contain one abstract method;
- lambda expressions can implement functional interfaces;
- `Comparable` defines a natural ordering;
- `Comparator` defines alternative orderings;
- interfaces reduce coupling and improve extensibility;
- abstract classes and interfaces solve different design problems.

33. Review Questions

1. What is an interface?
2. What is the purpose of an interface?
3. Which keyword is used when a class implements an interface?
4. Can an interface be instantiated directly?
5. Why must an implementing method usually be public?
6. Can a class implement more than one interface?
7. Does Java support multiple inheritance of classes?
8. How does Java support multiple-type inheritance?
9. Can an interface extend another interface?
10. Can one interface extend several interfaces?
11. What access and modifiers do interface fields implicitly have?
12. What is a default method?
13. Why were default methods introduced?
14. Can a class override a default method?
15. What happens if two interfaces provide the same default method?
16. How can a specific interface default method be called?
17. What is a static interface method?
18. Is a static interface method inherited as an instance method?
19. What is a private interface method?
20. What is the difference between an abstract class and an interface?
21. When is an abstract class preferable?
22. When is an interface preferable?
23. What is a functional interface?
24. What does `@FunctionalInterface` do?
25. Can a functional interface contain default methods?
26. What is `Comparable<T>`?
27. What is `Comparator<T>`?
28. What is the difference between `compareTo()` and `compare()`?
29. What is a marker interface?
30. Why are interfaces important in large Java APIs?
31. What does interface-based design mean?
32. What is interface segregation?

34. Programming Exercises

Beginner Level

1. Create a `Printable` interface and implement it in a `Report` class.
2. Create a `Drawable` interface and implement it in a `Circle` class.
3. Create a class that implements two interfaces.
4. Create two classes implementing the same interface and process them through interface references.
5. Create an interface with a constant and use it from a class.
6. Add a default method to an interface.
7. Override a default method in an implementing class.
8. Add a static utility method to an interface.
9. Create one interface that extends another.
10. Create an interface that extends two parent interfaces.

Intermediate Level

1. Create `Flyable` and `Swimmable` interfaces and implement both in a `Duck` class.
2. Create a `Payable` interface and implement it in two employee classes.
3. Create a `Shape` interface and implement it in three geometric classes.
4. Create two interfaces with conflicting default methods and resolve the conflict.
5. Create a functional interface for a mathematical operation and implement it with lambda expressions.
6. Implement `Comparable<Student>` using student name.
7. Sort students using a `Comparator<Student>` by GPA.
8. Create separate comparators for name, GPA, and age.
9. Create an interface-based notification system with email and SMS implementations.
10. Create an interface-based payment-processing system.

Advanced Level

1. Design a repository interface with both memory and file-like implementations.
2. Design a storage interface that can later be implemented by a database class.
3. Create a plug-in calculator in which every mathematical operation implements an `Operation` interface.
4. Create an authentication interface with at least two implementations.
5. Design a logging interface with console and file-like implementations.
6. Combine an abstract superclass with several interfaces in one hierarchy.
7. Create a service class that depends only on an interface and demonstrate changing implementations at runtime.
8. Design an interface hierarchy for a multimedia application.
9. Refactor a tightly coupled application so that the high-level logic depends on abstractions.
10. Build a modular mini-application using at least four interfaces and several implementations.

35. Homework

Task: Build an Interface-Based University Notification System

Create the following contracts:

Illustration

```
MessageSender
  |
  +-- EmailSender
  +-- SmsSender
  +-- ConsoleSender
```

Also create:

Illustration

```
NotificationService
```

Requirements:

1. Create an interface `MessageSender` with:
   - method `send(String message)`;
   - default method `senderInfo()`;
   - static method `validateMessage(String message)`.
2. Implement the interface in:
   - `EmailSender`;
   - `SmsSender`;
   - `ConsoleSender`.
3. Create `NotificationService` that receives a `MessageSender` through its constructor.
4. The service must not depend directly on a concrete sender class.
5. Demonstrate changing the sender implementation without changing the service logic.
6. Add another interface:

```
interface Loggable {
    void log(String message);
}
```

7. Make at least one sender implement both `MessageSender` and `Loggable`.
8. Create a functional interface `MessageFormatter` and implement it using at least two lambda expressions.
9. Use `Comparable` or `Comparator` to sort a small array of notification records by one

criterion of your choice.

Submission requirements:

- all Java source files;
- program output;
- a short diagram of the interfaces and implementing classes;
- explanation of where multiple interface implementation is used;
- explanation of where polymorphism is used;
- explanation of why `NotificationService` is loosely coupled;
- explanation of the difference between the abstract-class approach and the interface-based approach for this task.

36. Key Terms

Interface
A Java type that defines a behavioral contract.

`implements`
Keyword used when a class provides an interface implementation.

Multiple interface implementation
Implementing more than one interface in one class.

Interface inheritance
One interface extending one or more other interfaces.

Default method
An interface method with a default implementation.

Static interface method
A method that belongs to the interface itself.

Private interface method
A helper method visible only inside the interface.

Functional interface
An interface with exactly one abstract method.

Lambda expression
A concise implementation of a functional interface.

`Comparable`
Standard interface defining natural ordering.

`Comparator`
Standard interface defining external/custom ordering.

Marker interface
An interface used primarily to mark a class with a property.

Loose coupling
: Designing components so that they depend minimally on concrete implementations.

Interface segregation
: Preferring small focused interfaces over large general-purpose ones.