Creating Abstract Relationships with Interfaces

Jim Wilson

MOBILE SOLUTIONS DEVELOPER & ARCHITECT

@hedgehogjim   jwhh.com

Overview

The need for more than inheritance

Implementing an interface

Generic interfaces

Implementing multiple interfaces

Declaring an interface

Default methods

Effective software development

- Relies on reusability

Class inheritance is part of the solution

- Allows one class to leverage the implementation details of another

Inheritance has limitations

- A class directly extends only one class
- Constrains realistically available reusability

Interfaces

An interface defines a contract

Provides a list of operations

Does not focus on implementation details

Classes implement interfaces

Express conformance to contract

Provide necessary methods

An Interface Example

The Comparable interface demonstrates the value of interfaces

Challenge

Objects often need to be ordered

Rules of ordering different for each class

Comparable Interface

Provides a contract for ordering

Benefit

Enables broadly reusable sorting utilities

Need no knowledge of specific class

Exposes one method

- compareTo
- Receives a reference to another object

Method indicates relative relationship

- Indicates ordering between current object and received object

Return value

- Negative value: Current is ordered first
- Positive value: Received is ordered first
- Zero: Current and received are equal

Passenger.java

```java
class Passenger {
  private String name;
  private int memberLevel; // 3 (1st priority), 2, 1
  private int memberDays;
  public Passenger(String name, int memberLevel, int memberDays) {
    this.name = name;
    this.memberLevel = memberLevel;
    this.memberDays = memberDays;
  }
```

Passenger.java

```
class Passenger implements Comparable {
  private String name;
  private int memberLevel; // 3 (1st priority), 2, 1
  private int memberDays;
  public Passenger(String name, int memberLevel, int memberDays) {
    this.name = name;
    this.memberLevel = memberLevel;
    this.memberDays = memberDays;
  }
```

Passenger.java

```
  public int compareTo(Object o) {
    Passenger p = (Passenger) o;
    int returnValue = p.memberLevel - memberLevel;
    if(returnValue == 0)
        returnValue = p.memberDays - memberDays;
    return returnValue;
  }
} // other members elided
```

Main.java

```java
Passenger[] passengers = {
        new Passenger("Luisa", 1, 180),
        new Passenger("Jack", 1, 90),
        new Passenger("Ashanti", 3, 730),
        new Passenger("Harish", 2, 150),
};
Arrays.sort(passengers);

// passengers: Ashanti, Harish, Luisa, Jack
```

```
public interface Comparable<T> {
  int compareTo(T o);
}
```

Implementing a Generic Interface

Some interfaces support additional type information

- Use a concept known as generics
- Allows strong typing

Passenger.java

```java
class Passenger implements Comparable {
  public int compareTo(Object o) {
    Passenger p = (Passenger) o;
    int returnValue = p.memberLevel - memberLevel;
    if(returnValue == 0)
        returnValue = p.memberDays - memberDays;
    return returnValue;
  }
} // other members elided
```

Passenger.java

```java
class Passenger implements Comparable<Passenger> {
  public int compareTo(Object o) {
    Passenger p = (Passenger) o;
    int returnValue = p.memberLevel - memberLevel;
    if(returnValue == 0)
        returnValue = p.memberDays - memberDays;
    return returnValue;
  }
} // other members elided
```

Passenger.java

```
class Passenger implements Comparable<Passenger> {
  public int compareTo(Passenger p) {
    Passenger p = (Passenger) o;
    int returnValue = p.memberLevel - memberLevel;
    if(returnValue == 0)
        returnValue = p.memberDays - memberDays;
    return returnValue;
  }
} // other members elided
```

A class can implement multiple interfaces

- Separate interfaces with a comma
- No practical limit on the number of interfaces a class can implement

Flight.java

```java
public class Flight
    implements Comparable<Flight> {
  private int passengers;
  private int seats = 150;
  public ArrayList<Passenger> passengerList = new ArrayList<>();
  public int compareTo(Flight flight) { . . . }



} // other members elided
```

Flight.java

```java
public class Flight
    implements Comparable<Flight> {
  private int passengers;
  private int seats = 150;
  private ArrayList<Passenger> passengerList = new ArrayList<>();
  public int compareTo(Flight flight) { . . . }



} // other members elided
```

Flight.java

```java
public class Flight
    implements Comparable<Flight>, Iterable<Passenger> {
  private int passengers;
  private int seats = 150;
  private ArrayList<Passenger> passengerList = new ArrayList<>();
  public int compareTo(Flight flight) { . . . }
  public Iterator<Passenger> iterator() {
    return passengerList.iterator();
  }
}
```

Main.java

```
Flight f175 = new Flight(175);

f175.add1Passenger(new Passenger("Santiago"));
f175.add1Passenger(new Passenger("Julie"));
f175.add1Passenger(new Passenger("John"));
f175.add1Passenger(new Passenger("Geetha"));

for (Passenger p : f175)
    System.out.println(p.getName());

// Santiago
// Julie
// John
// Geetha
```

Iterable Interface

Main.java

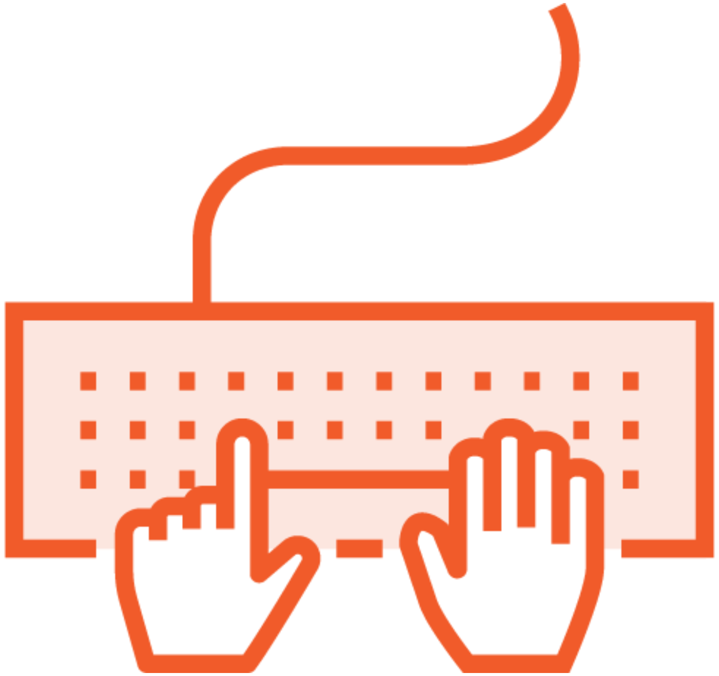
```
for (Passenger p : f175)
  System.out.println(p.getName());
```

Pseudo code

```
Iterable<Passenger> temp = f175;
Iterator<Passenger> iterator =
    temp.iterator();
while (iterator.hasNext()) {
  Passenger p = iteratator.next();
  System.out.println(p.getName());
}
```

Flight.java

```
public class Flight
    implements Comparable<Flight>, Iterable<Passenger> {
  private int passengers;
  private int seats = 150;
  private ArrayList<Passenger> passengerList = new ArrayList<>();
  public int compareTo(Flight flight) { . . . }
  public Iterator<Passenger> iterator() {
    return passengerList.iterator();
  }
}
```



Declaring an interface

- Similar to declaring a class
- Use the interface keyword

Declaring an Interface

Methods

Name, parameters, and return type

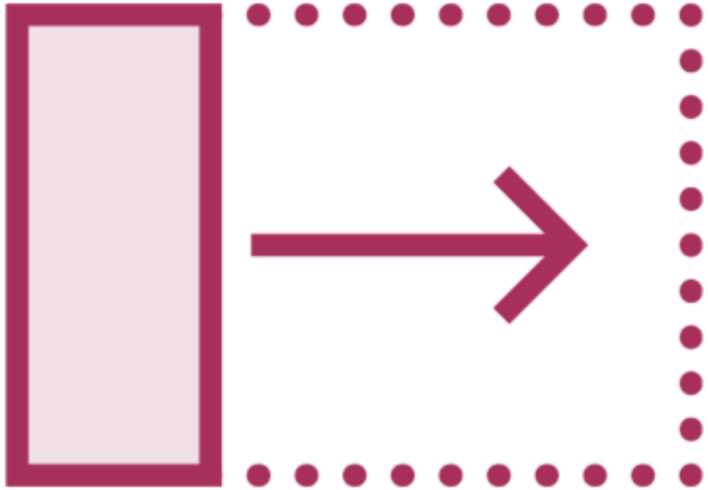
Implicitly public

Constant Fields

Typed named values

Implicitly public, final, and static

Extending Interfaces



An interface can extend another

Use extends keyword

Implementing a derived interface

Implies implementation of the base

Summary

An interface defines a contract

- Provides a list of operations
- Not focused on implementation details

Classes implement interfaces

- Express conformance to contract
- Provide necessary methods

Summary

Generic interfaces

- Allows stronger typing
- Allows specializing interface on a type

A class can implement multiple interfaces

- Separate with a comma
- Can implement as many as necessary

Summary

Declaring interfaces

- Use interface keyword
- Members implicitly public

Summary

Fields

- Constant values

Methods

- Name, parameters and return type
- Normally have no body

Default methods

- Have a body
- Allows adding methods to interface without breaking existing classes