

Interface

- ❖ Interface definition
- ❖ Interface implementation by classes
- ❖ Benefits of interfaces
- ❖ Implementation of multiple interface
- ❖ Java Collection Framework
- ❖ Sorting with Comparable<T> and Comparator<T>
- ❖ Default interface methods
- ❖ Static interface methods

Interfaces

- ❖ Interfaces is a way of describing what classes should do, without specifying how they should do it.
- ❖ An interface defines a set of methods.
- ❖ An interface declaration contains signatures, but no implementations

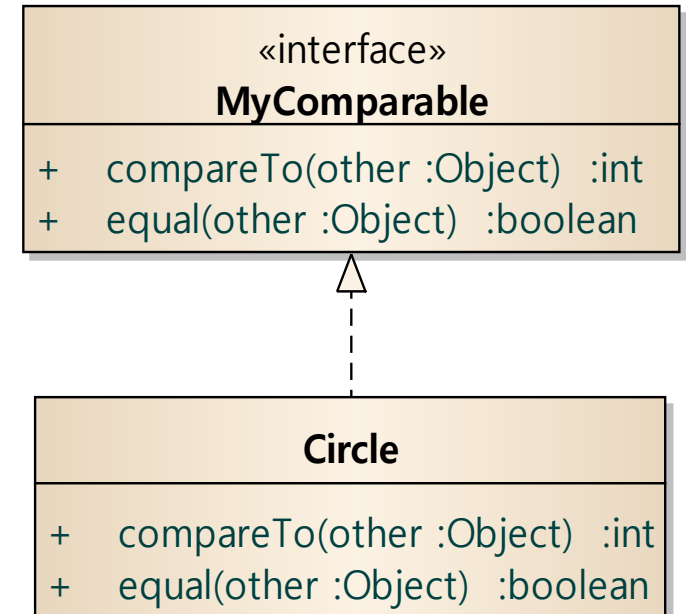
```
public interface MyComparable {  
    public int compareTo(Object other) ;  
    public boolean equal(Object other) ;  
}
```

«interface» MyComparable	
+	compareTo(other :Object) :int
+	equal(other :Object) :boolean

Interfaces

- ❖ A class can implement an interface. The class must implement all the methods declared in the interface

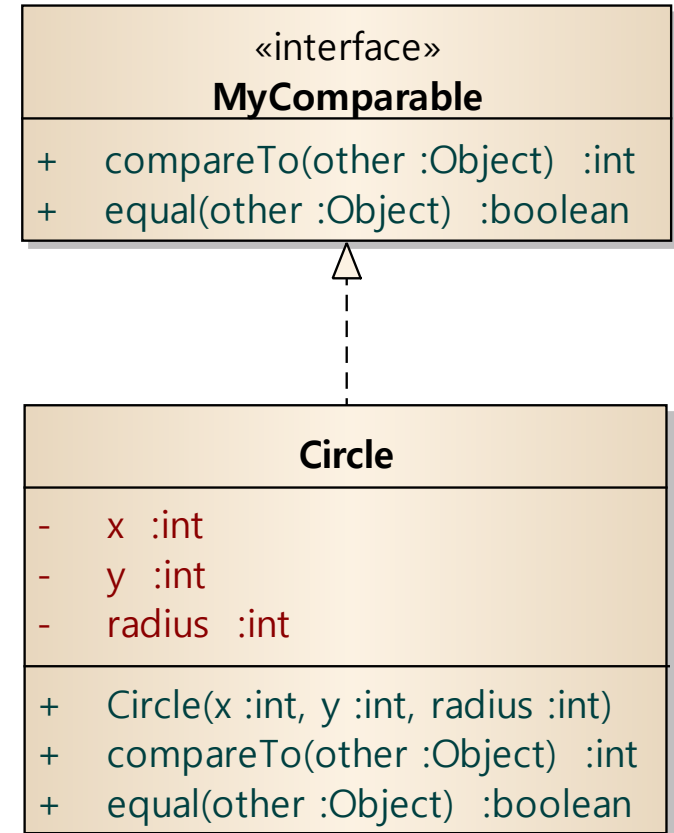
```
public class Circle implements MyComparable {  
    ...  
    public int compareTo(Object other) {  
        ...  
    }  
    public boolean equal(Object other) {  
        ...  
    }  
}
```



Interfaces

```
public class Circle implements MyComparable {
    private int x, y, radius ;

    public Circle(int x, int y, int radius) {
        this.x = x ; this.y = y ; this.radius = radius ;
    }
    public int compareTo(Object other) {
        if ( ! other instanceof Circle ) return -2 ;
        Circle otherCircle = (Circle) other ;
        int returnValue = 0 ;
        if ( radius < otherCircle.radius ) returnValue = -1 ;
        if ( radius == otherCircle.radius ) returnValue = 0 ;
        if ( radius > otherCircle.radius ) returnValue = 1 ;
        return returnValue ;
    }
    public boolean equal(Object other) {
        if ( ! other instanceof Circle ) return false ;
        Circle otherCircle = (Circle) other ;
        return x == otherCircle.x && y == otherCircle.y
            && radius == otherCircle.radius ;
    }
}
```



Interface based programming

- ❖ Interface variable can point to objects of a class that implements the interface

```
Circle c1 = new Circle(0, 0, 10) ;  
Circle c2 = new Circle(10, 10, 10) ;  
Circle c3 = new Circle(0, 0, 10) ;  
  
MyComparable[] list = new MyComparable[10] ;  
list[0] = c1 ;  
list[1] = c2 ;  
list[2] = c3 ;
```



List can point to Circle object.

Interface based programming

❖ Objects of a class can be accessed through its interface.

```
public class CircleTest {  
    public static void main(String[] args) {  
        Circle c1 = new Circle(0, 0, 10) ;  
        Circle c2 = new Circle(10, 10, 10) ;  
        Circle c3 = new Circle(0, 0, 10) ;  
  
        MyComparable[] list = {c1, c2, c3} ;  
  
        for ( int i = 0 ; i < list.length ; i ++ ) {  
            if ( list[0].compareTo(list[i]) < 0 )  
                System.out.println(list[0] + " has smaller size than " + list[i]) ;  
            if ( list[0].compareTo(list[i]) == 0 )  
                System.out.println(list[0] + " has the same size as " + list[i]) ;  
            if ( list[0].compareTo(list[i]) == 0 )  
                System.out.println(list[0] + " has the larger size than " + list[i]) ;  
        } // actually, Circle is a subclass of Object. So, toString() can be invoked.  
    }  
}
```

Each Circle object is accessed through MyComparable interface

Benefits of Interfaces

- ❖ Interfaces support generalized functions for different classes.
- ❖ Therefore, MySort can be used with any class which implements MyComparable interface

```
public class MySort {  
    public static void sort(MyComparable[] elements) {  
        for ( int i = 0 ; i < elements.length - 1; i ++ ) {  
            for ( int j = i + 1 ; j < elements.length ; j ++ ) {  
                if ( elements[i].compareTo(elements[j]) > 0 ) {  
                    MyComparable temp = elements[j] ;  
                    elements[j] = elements[i] ;  
                    elements[i] = temp ;  
                }  
            }  
        }  
    }  
}
```

Benefits of Interfaces

❖ MySort with Circle

```
public class CircleSortTest {  
    public static void main(String[] args) {  
        Circle c1 = new Circle(0, 0, 15) ;  
        Circle c2 = new Circle(10, 10, 10) ;  
        Circle c3 = new Circle(0, 0, 20) ;  
  
        MyComparable[] list = {c1, c2, c3} ;  
        MySort.sort(list) ;  
  
        for ( Object o : list ) // for (Circle o : list) is allowed?  
            System.out.println(o) ;  
    }  
}
```

```
Center: [ 10, 10], Radius: 10  
Center: [ 0, 0], Radius: 15  
Center: [ 0, 0], Radius: 20
```


Benefits of Interfaces

❖ MySort with Student

```
public class StudentSortTest {  
    public static void main(String[] args) {  
        Student s1 = new Student(1, "공부잘하는학생", 4.5F) ;  
        Student s2 = new Student(2, "공부잘못하는학생", 2.5F) ;  
  
        MyComparable[] list = {s1, s2} ;  
        MySort.sort(list) ;  
  
        for ( Object o : list ) System.out.println(o) ;  
    }  
}
```

ID:	2, Name:	공부잘못하는학생, GPA: 2.50
ID:	1, Name:	공부잘하는학생, GPA: 4.50

Student class

```
public class Student implements MyComparable {  
    private int studentID ;  
    private String name ;  
    private float gpa ;  
    public Student(int id, String name, float gpa) {  
        studentID = id ; this.name = name ; this.gpa = gpa ;  
    }  
    public int compareTo(Object other) {  
        Student otherStudent = (Student) other ;  
        int returnValue = 0 ;  
        if ( gpa < otherStudent.gpa ) returnValue = -1 ;  
        if ( gpa == otherStudent.gpa ) returnValue = 0 ;  
        if ( gpa > otherStudent.gpa ) returnValue = 1 ;  
        return returnValue ;  
    }  
    public boolean equal(Object other) { return studentID==((Student) other).studentID; }  
    public String toString() {  
        return String.format("ID: %5d, Name: %15s, GPA: %5.2f", studentID, name, gpa) ;  
    }  
}
```

Implementation of Multi-interfaces

- ❖ A class can implement two or more interfaces.

```
public interface AreaComputable {  
    public float getArea() ;  
}
```

```
public class Circle2 implements MyComparable, AreaComputable {  
    private int x, y ;  
    private int radius ;  
  
    public Circle2(int x, int y, int radius) {  
        this.x = x ; this.y = y ; this.radius = radius ;  
    }  
    public float getArea() { return (float) Math.PI * radius * radius ; }  
    public int compareTo(Object other) { ... }  
    public boolean equal(Object other) { ... }  
    public String toString() { ... }  
}
```

Triangle class

```
public class Triangle implements AreaComputable {  
    private int width, height ;  
    public Triangle(int width, int height) {  
        this.width = width ; this.height = height ;  
    }  
    public float getArea() { return (float) 0.5 * width * height ; }  
    public String toString() {  
        return String.format("Width: %5d, Height: %5d", width, height) ;  
    }  
}
```

Implementation of Multi-interfaces

```
public class AreaComputableTest {  
    public static void main(String[] args) {  
        Circle2 c1 = new Circle2(0, 0, 15) ;  
        Circle2 c2 = new Circle2(10, 10, 10) ;  
  
        Triangle t1 = new Triangle(10, 20) ;  
        Triangle t2 = new Triangle(20, 20) ;
```

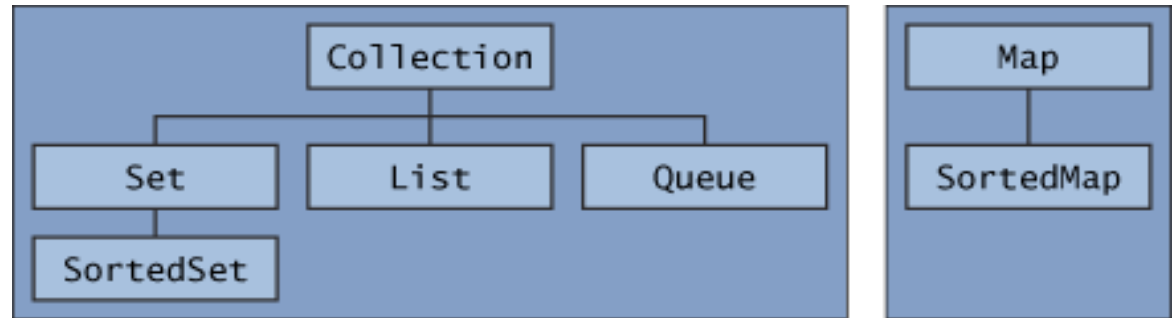
Center: [0, 0], Radius: 15	Area: 706.86
Center: [10, 10], Radius: 10	Area: 314.16
Width: 10, Height: 20	Area: 100.00
Width: 20, Height: 20	Area: 200.00
Total Area	
1321.02	

```
    AreaComputable[] list = {c1, c2, t1, t2} ;
```

```
    float totalArea = 0 ;  
    for ( AreaComputable elem: list) {  
        final float area = elem.getArea() ;  
        System.out.printf("%-40s Area: %10.2f%n", elem, area) ;  
        totalArea += area ;  
    }  
    System.out.printf("Total Area%n%10.2f%n", totalArea) ;  
}
```

Java Collection Framework

- ❖ Standard Library for Collection
- ❖ Need to import java.util.*
- ❖ Interfaces



- ❖ General implementations

Interfaces	Implementations				
	Hash table	Resizable array	Tree	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue				LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

- ❖ Refer to <http://docs.oracle.com/javase/tutorial/collections/index.html>
- ❖ Refer to http://www.tutorialspoint.com/java/java_collections.htm

Collection Example: List

```
import java.util.*;
public class ListExample {
    public static void main(String[] args) {
        List<String> names = new ArrayList<>() ; // or LinkedList<String>()

        // add
        names.add("Park") ;
        names.add("Kim") ;

        // toString
        System.out.println(names.toString()) ; // [Park, Kim]

        // add
        names.add(1, "Lee") ;

        // size, get
        for ( int i = 0 ; i < names.size() ; i ++ ) System.out.println(names.get(i)) ;
        // Park
        // Lee
        // Kim
    }
}
```

Collection Example: List

```
// remove
names.remove("Kim") ;
// indexOf
int foundIndex = names.indexOf("Kim") ;
if ( foundIndex == -1 )
    System.out.println("Kim not Found") ; // Kim not Found
else
    System.out.println("Kim Found") ;

// Iterator
Iterator<String> it = names.iterator() ;
while ( it.hasNext() ) System.out.println(it.next()) ;
// Park
// Lee
}
}
```


Collection Example: Set

```
import java.util.*;
public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<>(); // or TreeSet<String>()
        for (String a : args)
            if ( !s.add(a) ) System.out.println("Duplicate detected: " + a);

        System.out.println(s.size() + " distinct words: " + s);
    }
}
```

Now run the program.

```
% java FindDups i came i saw i left
```

The following output is produced.

```
Duplicate detected: i
```

```
Duplicate detected: i
```

```
4 distinct words: [i, left, saw, came]
```

Collection Example: Map

```
import java.util.*;
public class MapExample {
    public static void main(String[] args) {
        Map<String, Integer> cityPopulation = new HashMap<>() ;

        cityPopulation.put("Busan", 350) ;
        cityPopulation.put("Seoul", 1000) ;
        cityPopulation.put("Daejon", 150) ;

        System.out.println(cityPopulation) ; // {Seoul=1000, Busan=350, Daejon=150}

        if ( cityPopulation.containsKey("Daejon") )
            System.out.println(cityPopulation.get("Daejon")) ; // 150

        cityPopulation.remove("Daejon") ;

        Set<String> cities = cityPopulation.keySet() ;
        System.out.println(cities) ; // [Seoul, Busan]

        Collection<Integer> population = cityPopulation.values() ;
        System.out.println(population) ; // [1000, 350]
    }
}
```

Collection Example: Map

```
for( String key : cityPopulation.keySet() ){  
    System.out.println( String.format("키 : %s, 값 : %s", key, cityPopulation.get(key)) );  
}  
  
Iterator<String> keys = cityPopulation.keySet().iterator();  
while( keys.hasNext() ){  
    String key = keys.next();  
    System.out.println( String.format("키 : %s, 값 : %s", key, cityPopulation.get(key)) );  
}  
  
for( Map.Entry<String, Integer> elem : cityPopulation.entrySet() ){  
    System.out.println( String.format("키 : %s, 값 : %s", elem.getKey(), elem.getValue()) );  
}
```

```
}  
}
```

```
키 : Seoul, 값 : 1000  
키 : Busan, 값 : 350  
키 : Seoul, 값 : 1000  
키 : Busan, 값 : 350  
키 : Seoul, 값 : 1000  
키 : Busan, 값 : 350
```

Sorting with Comparable<T> and Comparator<T>

Sorting Array of Basic Types

```
public class BasicSortingMain {  
    public static void main(String[] args) {  
        int[] intArr = {5,9,1,10};  
        Arrays.sort(intArr);  
        System.out.println(Arrays.toString(intArr));  
  
        String[] strArr = {"A", "C", "B", "Z", "E"};  
        Arrays.sort(strArr);  
        System.out.println(Arrays.toString(strArr));  
  
        List<String> strList = new ArrayList<>();  
        strList.add("A");  
        strList.add("C");  
        strList.add("B");  
        strList.add("Z");  
        strList.add("E");  
        Collections.sort(strList);  
        for ( String str: strList ) System.out.print(" "+str);  
    }  
}
```

[1, 5, 9, 10]
[A, B, C, E, Z]
A B C E Z

Sorting With Comparable<T> Interface

- ❖ To sort an Object by its property, you have to make the Object implement the **Comparable** interface and override the **compareTo()** method

```
public class SortingObjectMain {  
    public static void main(String[] args) {  
        //sorting object array  
        Employee[] empArr = new Employee[4];  
        empArr[0] = new Employee(10, "Mikey", 25, 10000);  
        empArr[1] = new Employee(20, "Arun", 29, 20000);  
        empArr[2] = new Employee(5, "Lisa", 35, 5000);  
        empArr[3] = new Employee(1, "Pankaj", 32, 50000);  
  
        //sorting employees array using Comparable interface implementation  
        Arrays.sort(empArr);  
  
        System.out.println("Default Sorting of Employees list:\n  
        +Arrays.toString(empArr));  
    }  
}
```

Default Sorting of Employees list:
[[id=1, name=Pankaj, age=32,
salary=50000], [id=5, name=Lisa,
age=35, salary=5000], [id=10,
name=Mikey, age=25,
salary=10000], [id=20, name=Arun,
age=29, salary=20000]]

Employee should implement
Comparable interface

Class Employee implementing Comparable

```
public class Employee implements Comparable<Employee> {  
    private int id;  
    private String name;  
    private int age;  
    private long salary;  
  
    public Employee(int id, String name, int age, int salary) {  
        this.id = id;  
        this.name = name;  
        this.age = age;  
        this.salary = salary;  
    }  
    public int getId() { return id; }  
    public void setId(int id) { this.id = id; }  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
    public int getAge() { return age; }  
    public void setAge(int age) { this.age = age; }  
    public long getSalary() { return salary; }  
    public void setSalary(long salary) { this.salary = salary; }
```

Class Employee implementing Comparable

```
public class Employee implements Comparable<Employee> {  
    ...  
  
    @Override  
    //this is overridden to print the user friendly information about the Employee  
    public String toString() {  
        return "[id=" + this.id + ", name=" + this.name + ", age=" +  
            this.age + ", salary=" + this.salary + "];"  
    }  
  
    @Override  
    public int compareTo(Employee emp) {  
        //let's sort the employee based on id in ascending order  
        //returns a negative integer, zero, or a positive integer as this employee id  
        //is less than, equal to, or greater than the specified object.  
        return (this.id - emp.id);  
    }  
}
```


Sorting With Comparator<T> Interface

- ❖ The Comparable interface is only allow to sort a single property. To sort with multiple properties, you need **Comparator<T>**.

```
public class SortingObjectMain {  
    public static void main(String[] args) {  
        //sorting object array  
        Employee[] empArr = new Employee[4];  
        ...  
  
        //sort employees array using Comparator by Salary  
        Arrays.sort(empArr, Employee.SalaryComparator);  
        System.out.println("Employees list sorted by Salary:\n"  
            + Arrays.toString(empArr));  
  
        //sort employees array using Comparator by Age  
        Arrays.sort(empArr, Employee.AgeComparator);  
        System.out.println("Employees list sorted by Age:\n"  
            + Arrays.toString(empArr));  
  
        //sort employees array using Comparator by Name  
        Arrays.sort(empArr, Employee.NameComparator);  
        System.out.println("Employees list sorted by Name:\n"  
            + Arrays.toString(empArr));  
    }  
}
```

Employees list sorted by Salary:

[[id=5, name=Lisa, age=35, salary=5000], [id=10, name=Mikey, age=25, salary=10000], [id=20, name=Arun, age=29, salary=20000], [id=1, name=Pankaj, age=32, salary=50000]]

Employees list sorted by Age:

[[id=10, name=Mikey, age=25, salary=10000], [id=20, name=Arun, age=29, salary=20000], [id=1, name=Pankaj, age=32, salary=50000], [id=5, name=Lisa, age=35, salary=5000]]

Employees list sorted by Name:

[[id=20, name=Arun, age=29, salary=20000], [id=5, name=Lisa, age=35, salary=5000], [id=10, name=Mikey, age=25, salary=10000], [id=1, name=Pankaj, age=32, salary=50000]]

Class Employee having Comparators

```
public class Employee {  
    private int id;  
    private String name;  
    private int age;  
    private long salary;  
  
    public Employee(int id, String name, int age, int salary) {  
        this.id = id;  
        this.name = name;  
        this.age = age;  
        this.salary = salary;  
    }  
    public int getId() { return id; }  
    public void setId(int id) { this.id = id; }  
    public String getName() { return name; }  
    public void setName(String name) { this.name = name; }  
    public int getAge() { return age; }  
    public void setAge(int age) { this.age = age; }  
    public long getSalary() { return salary; }  
    public void setSalary(long salary) { this.salary = salary; }  
}
```

Class Employee having Comparators

```
public class Employee {  
    ...  
public int compareTo(Employee emp) { return (this.id - emp.id); }  
    public static Comparator<Employee> SalaryComparator  
    = new Comparator<Employee>() {  
        public int compare(Employee e1, Employee e2) {  
            return (int) (e1.getSalary() - e2.getSalary());  
        }  
    };  
    public static Comparator<Employee> AgeComparator  
    = new Comparator<Employee>() {  
        public int compare(Employee e1, Employee e2) {  
            return e1.getAge() - e2.getAge();  
        }  
    };  
    public static Comparator<Employee> NameComparator  
    = new Comparator<Employee>() {  
        public int compare(Employee e1, Employee e2) {  
            return e1.getName().compareTo(e2.getName());  
        }  
    };  
}
```

DEFAULT INTERFACE METHOD

STATIC INTERFACE METHOD

Default Interface Method

Since Java 8(March 2014)

❖ Default interface methods

- They are declared with the **default** keyword at the beginning of the method signature, and they provide an implementation

```
public interface MyInterface {  
  
    // regular interface methods  
  
    public default void defaultMethod() {  
        // default method implementation  
    }  
}
```

- ❖ They allow us to add new methods to an interface that are automatically available in the implementations.
- ❖ Thus, there's no need to modify the implementing classes

Default Interface Method

```
public interface Vehicle {  
    public String getBrand();  
    public String speedUp();  
    public String slowDown();  
    public default String turnAlarmOn() {  
        return "Turning the vehicle alarm on.";  
    }  
    public default String turnAlarmOff() {  
        return "Turning the vehicle alarm off.";  
    }  
}
```

Default Interface Method

```
public class Car implements Vehicle {  
    private String brand;  
  
    public Car(String brand) {  
        this.brand = brand;  
    }  
  
    @Override  
    public String getBrand() {  
        return brand;  
    }  
  
    @Override  
    public String speedUp() {  
        return "The car is speeding up.";  
    }  
  
    @Override  
    public String slowDown() {  
        return "The car is slowing down.";  
    }  
}
```

```
public static void main(String[] args) {  
    Vehicle car = new Car("BMW");  
    System.out.println(car.getBrand());  
    System.out.println(car.speedUp());  
    System.out.println(car.slowDown());  
    System.out.println(car.turnAlarmOn());  
    System.out.println(car.turnAlarmOff());  
}
```

Static Interface Method

- ❖ Java 8 allows us to define and implement static methods in interfaces

```
public interface Vehicle {  
  
    // regular / default interface methods  
  
    public static int getHorsePower(int rpm, int torque) {  
        return (rpm * torque) / 5252;  
    }  
}
```

```
Vehicle.getHorsePower(2500, 480);
```

- ❖ A static method can be invoked within other static and default methods
- ❖ Static methods in interfaces make possible to group related utility methods, without having to create artificial utility classes

Static Interface Method

- ❖ Up to now, it has been common to place static methods in companion classes; you find pairs of interfaces and utility classes such as `Collection/Collections`, `Array/Arrays`, or `Path/Paths`
- ❖ `Paths` class only has a couple of factory methods. You can construct a path to a file or directory from a sequence of strings, such as `Paths.get("jdk1.8.0", "jre", "bin")`.
- ❖ In Java SE 8, one could have added this method to the `Path` interface:

```
public interface Path {  
    public static Path get(String first, String... more) {  
        return FileSystems.getDefault().getPath(first, more);  
    }  
    ...  
}
```

- ❖ It is unlikely that the Java library will be refactored in this way, but when you implement your own interfaces, there is no longer a reason to provide a separate companion class for utility methods

Interface TimeClient

```
import java.time.DateTimeException;
import java.time.LocalDateTime;
import java.time.ZoneId;
import java.time.ZonedDateTime;
public interface TimeClient {
    public void setTime(int hour, int minute, int second);
    public void setDate(int day, int month, int year);
    public void setDateAndTime(int day, int month, int year,
        int hour, int minute, int second);
    public LocalDateTime getLocalDateTime();

    public static ZoneId getZoneId (String zoneString) {
        try {
            return ZoneId.of(zoneString);
        } catch (DateTimeException e) {
            System.err.println("Invalid time zone: " + zoneString +
                "; using default time zone instead.");
            return ZoneId.systemDefault();
        }
    }

    public default ZonedDateTime getZonedDateTime(String zoneString) {
        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
    } // similar to template method
}
```

Class SimpleTimeClient

```
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.time.LocalTime;

public class SimpleTimeClient implements TimeClient {
    private LocalDateTime dateAndTime;
    public SimpleTimeClient() {
        dateAndTime = LocalDateTime.now();
    }
    public void setTime(int hour, int minute, int second) {
        LocalDate currentDate = LocalDate.from(dateAndTime);
        LocalTime timeToSet = LocalTime.of(hour, minute, second);
        dateAndTime = LocalDateTime.of(currentDate, timeToSet);
    }
    public void setDate(int day, int month, int year) {
        LocalDate dateToSet = LocalDate.of(day, month, year);
        LocalTime currentTime = LocalTime.from(dateAndTime);
        dateAndTime = LocalDateTime.of(dateToSet, currentTime);
    }
}
```

Class SimpleTimeClient

```
public void setDateAndTime(int day, int month, int year,
    int hour, int minute, int second) {
    LocalDate dateToSet = LocalDate.of(day, month, year);
    LocalTime timeToSet = LocalTime.of(hour, minute, second);
    dateAndTime = LocalDateTime.of(dateToSet, timeToSet);
}
public LocalDateTime getLocalDateTime() {
    return dateAndTime;
}
public String toString() {
    return dateAndTime.toString();
}

public static void main(String... args) {
    TimeClient myTimeClient = new SimpleTimeClient();
    System.out.println(myTimeClient.toString());
}
}
```

Class TestSimpleTimeClient

```
public class TestSimpleTimeClient {  
    public static void main(String... args) {  
        TimeClient myTimeClient = new SimpleTimeClient();  
  
        System.out.println("Current time: " + myTimeClient.toString());  
        System.out.println("Time in Seoul: " +  
            myTimeClient.getZonedDateTime("Asia/Seoul").toString());  
    }  
}
```

Extending Interfaces That Contain Default Methods

- ❖ When you extend an interface that contains a default method, you can do the following:
 - 1) Not mention the default method at all, which lets your extended interface inherit the default method.

```
public interface AnotherTimeClient extends TimeClient { }
```

- 2) Redeclare the default method, which makes it abstract.

```
public interface AbstractZoneTimeClient extends TimeClient {  
    public ZonedDateTime getZonedDateTime(String zoneString);  
}
```

Extending Interfaces That Contain Default Methods

- 3) Redefine the default method, which overrides it.

```
public interface HandleInvalidTimeZoneClient extends TimeClient {  
    public default ZonedDateTime getZonedDateTime(String zoneString) {  
        try {  
            return ZonedDateTime.of(getLocalDateTime(), ZonedDateTime.of(zoneString));  
        } catch (DateTimeException e) {  
            System.err.println("Invalid zone ID: " + zoneString +  
                "; using the default time zone instead.");  
            return ZonedDateTime.of(getLocalDateTime(), ZonedDateTime.systemDefault());  
        }  
    }  
}
```