

2023/2024(1)  
EF234302 Object Oriented Programming

Lecture #5b

# Interface

Misbakhul Munir **IRFAN SUBAKTI**

司馬伊凡

Мисбакхул Мунир **Ирфан Субакти**

# Method: Multiple versions?

- In many situations it may be the case that we have many objects that are very similar, and want to be able to apply very similar methods to them. The problem is, however, that the finer details in these operations (fetching one variable from one object type compared to another) means that we have to define multiple versions of the same method to handle each case.
- Take, for example, a method to compute the average balance of an array of `BankAccount` objects:

```
public static double average(BankAccount objects[]) {  
    if (objects.length == 0) { return 0; }  
    double sum = 0.0;  
    // Loops through each object in array  
    for(BankAccount obj : objects) {  
        sum = sum + obj.getBalance();  
    }  
    return sum / objects.length;  
}
```

# Method: Multiple versions? (continued)

- This is a simple method. Now suppose we have a list of (Country) objects and we want to calculate the average area:

```
public static double average(Country objects[]) {  
    if (objects.length == 0) { return 0; }  
    double sum = 0.0;  
    // Loops through each object in array  
    for(Country obj : objects) {  
        sum = sum + obj.getArea();  
    }  
    return sum / objects.length;  
}
```

- As we can see, the two methods are virtually identical, except for the object types and the getter name.
- This is where interfaces come in. An interface is essentially the instructions for making a new object. An interface is created using the `interface` keyword:

```
public interface Measurable {...}
```

- An interface will contain the names, return types and parameters of any methods that must be featured in a class using the `interface`, for example:

```
public interface Measurable {  
    public double getMeasure();  
}
```

- Notice that the method is missing a body, only has *function header*. A method in an interface must be `abstract`, i.e., it contains *no code*.

# Implementing an Interface

- Now we have our interface, we can *“implement”* it by using the `implements` keyword. This comes in the class declaration, and a class can implement **many interfaces**. What if we take the `Measurable` example from above in the case of the `BankAccount` and `Country` classes? We can now make the `BankAccount` and `Country` classes implement `Measurable`:

```
// BankAccount Class
public class BankAccount implements Measurable {
    public double getMeasure() {
        return balance;
    }
}
```

```
// Country Class
public class Country implements Measurable {
    public double getMeasure() {
        return area;
    }
}
```

# Implementation (continued)

- We can now rewrite our `average()` method to handle both of these classes:

```
public static double average(Measurable objects[]) {
    if (objects.length == 0) { return 0; }
    double sum = 0;
    // Loops through each object in array
    for (Measurable obj : objects) {
        sum = sum + obj.getMeasure();
    }
    return sum / objects.length;
}
```

- We can then use these as follows:

```
public static void main(String[] args) {
    // One way
    Measurable accounts[] = new Measurable[3];
    accounts[0] = new BankAccount("01", "Aleksandra", 100);
    accounts[1] = new BankAccount("02", "Natasha", 150);
    accounts[2] = new BankAccount("03", "Sergei", 125);
    System.out.println("Average balance is: " + average(accounts));
    // Second way
    BankAccount b1 = new BankAccount("01", "Aleksandra", 100);
    BankAccount b2 = new BankAccount("02", "Natasha", 150);
    BankAccount accounts2[] = {b1, b2};
    System.out.println("Average balance is: " + average(accounts2));
}
```

# Implementation (continued)

- There are a few important points here. While we can define variables that have a type of an interface:

```
Measurable x;
```

- There is no constructor for an interface, i.e., the following will result in an error.

```
Measurable x = new Measurable(); // Error
```

- We can, however, make an object that is defined as Measurable, but is constructed as an object that implements it:

```
Measurable y = new BankAccount("01", "Aleksandra", 100);
```

# Interface: Usage

- One example of use for an interface is in an industrial setting.
- Imagine a piece of image editing software that provides a set of built in methods which apply some operations.
- The code to these is kept secret, however, as the algorithms are trade secrets.
- The program allows the user to create their own plugins using Java objects.
- The user still need to be able to use these built in methods, without viewing the source code.
- All that needs to be done by the software publishers is to provide an interface for user-built objects.
- This interface can then be used to allow any custom user object to be passed to the secret methods.

# Interface: Database project example

- Scenario
  - Two methods of representing a table (here called a “database”)
  - Client wants to choose one or the other, based on information determined when it is first run



# LinearDB & TreeDB classes

- Client decides at the start which representation to use, then has an `if` whenever it does a database operation

```
LinearDB.java x
1 package dbrep;
2 public class LinearDB {
3     private int k;
4     // Represent data using unsorted array
5     int keys[];
6     void addKey (int k) {
7         System.out.println("LinearDB: " +
8             "Adding key process");
9         this.k = k;
10    }
11    boolean search (int k) {
12        // Assume it's found for k = 9
13        boolean result = (k == 9) ? true
14                        : false;
15        System.out.println("LinearDB: " +
16            "Searching process");
17        return result;
18    }
19 }
```

```
LinearDB.java TreeDB.java x
1 package dbrep;
2 public class TreeDB {
3     private int k;
4     // Represent data using binary search tree
5     BinarySearchTree data;
6     void addKey (int k) {
7         System.out.println("TreeDB: " +
8             "Adding key process");
9         this.k = k;
10    }
11    boolean search (int k) {
12        // Assume it's found for k = 9
13        boolean result = (k == 9) ? true
14                        : false;
15        System.out.println("TreeDB: " +
16            "Searching process");
17        return result;
18    }
19 }
```

```
LinearDB.java TreeDB.java BinarySearchTree.java x
1 package dbrep;
2 public class BinarySearchTree {
3 }
```

# DBClient class

- Thus, every use of database operation must be enclosed in test to make sure correct variable is used

```
LinearDB.java TreeDB.java BinarySearchTree.java DBClient.java x
1 package dbrep;
2 public class DBClient {
3     static LinearDB ldb;
4     static TreeDB tdb;
5     static boolean useLinearDB = true;
6     public static void main(String[] args) {
7         // Decide which rep. to use
8         // String decision = "Linear Rep";
9         String decision = "Tree Rep";
10        // It turns that linear rep has been chosen
11        if (decision.equals("Linear Rep")) {
12            ldb = new LinearDB();
13        } else { // Otherwise, tree rep will be used
14            useLinearDB = false;
15            tdb = new TreeDB();
16        }
17        // Add key k1
18        int k1 = 13;
19        if (useLinearDB) {
20            ldb.addKey(k1);
21        } else {
22            tdb.addKey(k1);
23        }
24        // Search for key k2
25        int k2 = 9;
26        boolean result = false;
27        result = useLinearDB ? ldb.search(k2)
28                    : tdb.search(k2);
29        System.out.println(result);
30    }
31 }
```

# Database project using Interface

- Recap
  - Interfaces are a type of Java component which is like a class, but contains only *function headers*, not definitions, *no code*
  - It is used to declare the set of operations that an object may have
- Alternative to this structure can be obtained by using DBops interface, as follows:

```
DBops.java ×  
1 package dbrepInterface;  
2 public interface DBops {  
3     void addKey(int k);  
4     boolean search(int k);  
5 }
```

# Using Interface

- Change LinearDB and TreeDB as follows:
  - In header, add `implements DBops`
  - Declare `addKey` and `search` as `public` (no other changes needed)

```
LinearDB.java ×
1 package dbrepInterface;
2 public class LinearDB implements DBops {
3     private int k;
4     // Represent data using unsorted array
5     int keys[];
6     public void addKey (int k) {
7         System.out.println("LinearDB: " +
8             "Adding key process");
9         this.k = k;
10    }
11    public boolean search (int k) {
12        // Assume it's found for k = 9
13        boolean result = (k == 9) ? true
14                        : false;
15        System.out.println("LinearDB: " +
16            "Searching process");
17        return result;
18    }
19 }
```

```
DBops.java LinearDB.java TreeDB.java ×
1 package dbrepInterface;
2 public class TreeDB implements DBops {
3     private int k;
4     // Represent data using binary search tree
5     BinarySearchTree data;
6     public void addKey (int k) {
7         System.out.println("TreeDB: " +
8             "Adding key process");
9         this.k = k;
10    }
11    public boolean search (int k) {
12        // Assume it's found for k = 9
13        boolean result = (k == 9) ? true
14                        : false;
15        System.out.println("TreeDB: " +
16            "Searching process");
17        return result;
18    }
19 }
```

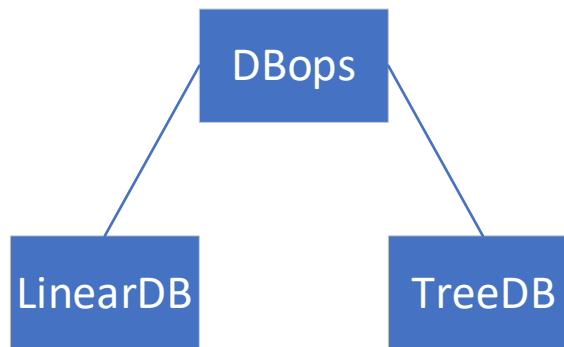
# Update DBClient class

- Use `DBops` as a type, and declare a variable, say `db`, of that type. `db` can contain a reference either to a `LinearDB` or a `TreeDB` object
  - Interfaces can be used as **types** to declare variables
- Assign appropriate type of object to `db`
- Use ordinary instance method call syntax with `db` as the receiver

```
DBops.java  LinearDB.java  TreeDB.java  DBClient.java ×
1 package dbrepInterface;
2 public class DBClient {
3     static DBops db;
4     // static LinearDB ldb;
5     // static TreeDB tdb;
6     static boolean useLinearDB = true;
7     public static void main(String[] args) {
8         // Decide which rep. to use
9         // String decision = "Linear Rep";
10        String decision = "Tree Rep";
11        // It turns that linear rep has been chosen
12        if (decision.equals("Linear Rep")) {
13            db = new LinearDB();
14        } else { // Otherwise, tree rep will be used
15            useLinearDB = false;
16            db = new TreeDB();
17        }
18        // Add key k1
19        int k1 = 13;
20        db.addKey(k1);
21        // Search for key k2
22        int k2 = 9;
23        boolean result = false;
24        result = db.search(k2);
25        System.out.println(result);
26    }
27 }
```

# Type hierarchies

- Objects of type `LinearDB` and `TreeDB` (“subtypes”) can be regarded as being of type `DBops` (the “supertype”)
- During the assignment and parameter-passing, such conversion from subtypes to supertypes happens automatically



# Interfaces in Java API

- `java.lang` have many interfaces
  - `Comparable`,
  - `Runnable`, ...

Interface Summary	
Interface	Description
<code>Appendable</code>	An object to which <code>char</code> sequences and values can be appended.
<code>AutoCloseable</code>	An object that may hold resources (such as file or socket handles) until it is closed.
<code>CharSequence</code>	A <code>CharSequence</code> is a readable sequence of <code>char</code> values.
<code>Cloneable</code>	A class implements the <code>Cloneable</code> interface to indicate to the <code>Object.clone()</code> method that it is legal for that method to make a field-for-field copy of instances of that class.
<code>Comparable&lt;T&gt;</code>	This interface imposes a total ordering on the objects of each class that implements it.
<code>Iterable&lt;T&gt;</code>	Implementing this interface allows an object to be the target of the "for-each loop" statement.
<code>Readable</code>	A <code>Readable</code> is a source of characters.
<code>Runnable</code>	The <code>Runnable</code> interface should be implemented by any class whose instances are intended to be executed by a thread.
<code>Thread.UncaughtExceptionHandler</code>	Interface for handlers invoked when a <code>Thread</code> abruptly terminates due to an uncaught exception.

# Interfaces in Java API (continued)

- `java.util` have many interfaces
  - `Collection`,  
`Comparator`, `List`,  
`Map`, ...

Interface Summary	
Interface	Description
<code>Collection&lt;E&gt;</code>	The root interface in the <i>collection hierarchy</i> .
<code>Comparator&lt;T&gt;</code>	A comparison function, which imposes a <i>total ordering</i> on some collection of objects.
<code>Deque&lt;E&gt;</code>	A linear collection that supports element insertion and removal at both ends.
<code>Enumeration&lt;E&gt;</code>	An object that implements the <code>Enumeration</code> interface generates a series of elements, one at a time.
<code>EventListener</code>	A tagging interface that all event listener interfaces must extend.
<code>Formattable</code>	The <code>Formattable</code> interface must be implemented by any class that needs to perform custom formatting using the 's' conversion specifier of <code>Formatter</code> .
<code>Iterator&lt;E&gt;</code>	An iterator over a collection.
<code>List&lt;E&gt;</code>	An ordered collection (also known as a <i>sequence</i> ).
<code>ListIterator&lt;E&gt;</code>	An iterator for lists that allows the programmer to traverse the list in either direction, modify the list during iteration, and obtain the iterator's current position in the list.
<code>Map&lt;K,V&gt;</code>	An object that maps keys to values.
<code>Map.Entry&lt;K,V&gt;</code>	A map entry (key-value pair).
<code>NavigableMap&lt;K,V&gt;</code>	A <code>SortedMap</code> extended with navigation methods returning the closest matches for given search targets.
<code>NavigableSet&lt;E&gt;</code>	A <code>SortedSet</code> extended with navigation methods reporting closest matches for given search targets.
<code>Observer</code>	A class can implement the <code>Observer</code> interface when it wants to be informed of changes in observable objects.
<code>PrimitiveIterator&lt;T,T_CONS&gt;</code>	A base type for primitive specializations of <code>Iterator</code> .
<code>PrimitiveIterator.OfDouble</code>	An <code>Iterator</code> specialized for double values.
<code>PrimitiveIterator.OfInt</code>	An <code>Iterator</code> specialized for int values.
<code>PrimitiveIterator.OfLong</code>	An <code>Iterator</code> specialized for long values.
<code>Queue&lt;E&gt;</code>	A collection designed for holding elements prior to processing.
<code>RandomAccess</code>	Marker interface used by <code>List</code> implementations to indicate that they support fast (generally constant time) random access.
<code>Set&lt;E&gt;</code>	A collection that contains no duplicate elements.
<code>SortedMap&lt;K,V&gt;</code>	A <code>Map</code> that further provides a <i>total ordering</i> on its keys.
<code>SortedSet&lt;E&gt;</code>	A <code>Set</code> that further provides a <i>total ordering</i> on its elements.
<code>Splitter&lt;T&gt;</code>	An object for traversing and partitioning elements of a source.
<code>Splitter.OfDouble</code>	A <code>Splitter</code> specialized for double values.
<code>Splitter.OfInt</code>	A <code>Splitter</code> specialized for int values.
<code>Splitter.OfLong</code>	A <code>Splitter</code> specialized for long values.
<code>Splitter.OfPrimitive&lt;T,T_CONS,T_SPLITR extends Splitter.OfPrimitive&lt;T,T_CONS,T_SPLITR&gt;&gt;</code>	A <code>Splitter</code> specialized for primitive values.



# Comparable

- Most Collection classes have a `sort` method
- Sorting involves comparing elements
- Comparison method to `sort` via the `Comparable` interface

```
public interface Comparable {  
    public int compareTo(Object o);  
}
```

# Comparable: Example

- Suppose we have an array of `names`, where each name consists of a last name and a first name

```
public class Name implements Comparable {
    String lastName, firstName;
    public Name(String lastName, String firstName) {
        ...
    }
    int compareTo(Object o) {
        ...
    }
}
```

# Comparable: Example

```
Name.java x
1 public class Name implements Comparable {
2     private String lastName, firstName;
3     public Name(String lastName, String firstName) {
4         this.lastName = lastName;
5         this.firstName = firstName;
6     }
7     @Override
8     public int compareTo(Object other) {
9         Name o = (Name) other;
10        int result = Integer.MIN_VALUE;
11        switch (lastName.compareTo(o.lastName)) {
12            case -1:
13                result = -1;
14                break;
15            case 1:
16                result = 1;
17                break;
18            case 0:
19                result = (firstName.compareTo(o.firstName));
20                break;
21        }
22        return result;
23    }
24    @Override
25    public String toString() {
26        return "[" + lastName + ", " + firstName + "];"
27    }
28 }
```

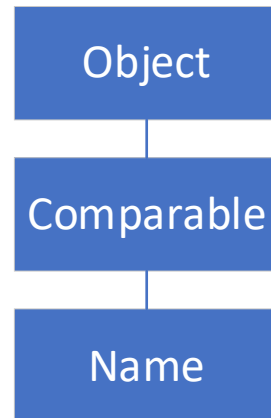
```
Name.java Main.java x
1 public class Main {
2     public static void main(String[] args) {
3         Name n1 = new Name("Zonk", "Godot");
4         Name n2 = new Name("Bogdanov", "Alexei");
5         Name n3 = new Name("Ace", "Portgas D.");
6         Name names[] = {n1, n2, n3};
7         System.out.println("Original order:");
8         for (Name n : names) {
9             System.out.println(n);
10        }
11        java.util.Arrays.sort(names);
12        System.out.println("\nAfter sorting:");
13        for (Name n : names) {
14            System.out.println(n);
15        }
16    }
17 }
18
19 }
```

```
Original order:
[Zonk, Godot]
[Bogdanov, Alexei]
[Ace, Portgas D.]

After sorting:
[Ace, Portgas D.]
[Bogdanov, Alexei]
[Zonk, Godot]
```

# Type hierarchies

- Subtype-to-supertype conversion happens automatically
- However, supertype-to-subtype conversion requires an explicit cast
  - Down-casting



# Interfaces in AWT package

- Interfaces in java.lang.awt

```
interface ActionListener {  
    void actionPerformed(ActionEvent e);  
}
```

- **Button** objects, among others, can have action listeners:

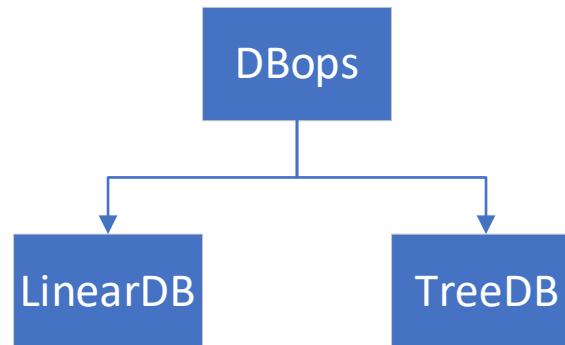
```
addActionListener(ActionListener)  
removeActionListener(ActionListener)
```

# Interfaces in AWT package (continued)

- Any object that implements the `ActionListener` interface can add itself as a listener for a button
- Suppose we had an animation with bouncing balls, pendulums, rotating spirals, etc. All of them can be listeners for a **Stop** button

# Interfaces: Pros and Cons

- **Pro:** Classes can implement *any number* of interfaces
- **Con:** Interfaces contain no code, only *declarations* of methods



# Code sharing

- Suppose we want both `LinearDB` and `TreeDB` to have the following extra methods:

```
void addSeveral (int keys[])  
boolean findOneOf (int keys[])
```

- Both with have similar-looking code. Should we repeat it in both classes?