2021/2022(1)
IF184301 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*. An 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

```
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

```
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

```
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

```
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

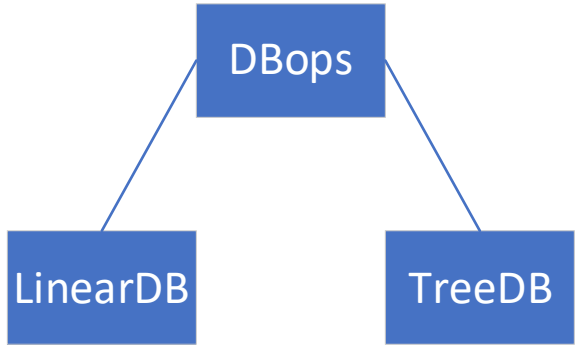
- Use `DBops` as a type, and declare a variable, say `db`, of that type. `db` can contain a reference either to a `LinearDB` or a `TreeDB` object
  - Interfaces can be used as **types** to declare variables
- Assign appropriate type of object to `db`
- Use ordinary instance method call syntax with `db` 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
Appendable	An object to which `char` sequences and values can be appended.
AutoCloseable	An object that may hold resources (such as file or socket handles) until it is closed.
CharSequence	A `CharSequence` is a readable sequence of `char` values.
Cloneable	A class implements the `Cloneable` interface to indicate to the `Object.clone()` method that it is legal for that method to make a field-for-field copy of instances of that class.
Comparable<T>	This interface imposes a total ordering on the objects of each class that implements it.
Iterable<T>	Implementing this interface allows an object to be the target of the "for-each loop" statement.
Readable	A `Readable` is a source of characters.
Runnable	The `Runnable` interface should be implemented by any class whose instances are intended to be executed by a thread.
Thread.UncaughtExceptionHandler	Interface for handlers invoked when a `Thread` 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
Collection<E>	The root interface in the *collection hierarchy*.
Comparator<T>	A comparison function, which imposes a *total ordering* on some collection of objects.
Deque<E>	A linear collection that supports element insertion and removal at both ends.
Enumeration<E>	An object that implements the Enumeration interface generates a series of elements, one at a time.
EventListener	A tagging interface that all event listener interfaces must extend.
Formattable	The `Formattable` interface must be implemented by any class that needs to perform custom formatting using the `'s'` conversion specifier of `Formatter`.
Iterator<E>	An iterator over a collection.
List<E>	An ordered collection (also known as a *sequence*).
ListIterator<E>	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.
Map<K,V>	An object that maps keys to values.
Map.Entry<K,V>	A map entry (key-value pair).
NavigableMap<K,V>	A `SortedMap` extended with navigation methods returning the closest matches for given search targets.
NavigableSet<E>	A `SortedSet` extended with navigation methods reporting closest matches for given search targets.
Observer	A class can implement the `Observer` interface when it wants to be informed of changes in observable objects.
PrimitiveIterator<T,T_CONS>	A base type for primitive specializations of `Iterator`.
PrimitiveIterator.OfDouble	An Iterator specialized for `double` values.
PrimitiveIterator.OfInt	An Iterator specialized for `int` values.
PrimitiveIterator.OfLong	An Iterator specialized for `long` values.
Queue<E>	A collection designed for holding elements prior to processing.
RandomAccess	Marker interface used by `List` implementations to indicate that they support fast (generally constant time) random access.
Set<E>	A collection that contains no duplicate elements.
SortedMap<K,V>	A `Map` that further provides a *total ordering* on its keys.
SortedSet<E>	A `Set` that further provides a *total ordering* on its elements.
Spliterator<T>	An object for traversing and partitioning elements of a source.
Spliterator.OfDouble	A Spliterator specialized for `double` values.
Spliterator.OfInt	A Spliterator specialized for `int` values.
Spliterator.OfLong	A Spliterator specialized for `long` values.
Spliterator.OfPrimitive<T,T_CONS,T_SPLITR extends Spliterator.OfPrimitive<T,T_CONS,T_SPLITR>>	A Spliterator 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

```java
public class Name implements Comparable {
    private String lastName, firstName;
    public Name(String lastName, String firstName) {
        this.lastName = lastName;
        this.firstName = firstName;
    }
    @Override
    public int compareTo(Object other) {
        Name o = (Name) other;
        int result = Integer.MIN_VALUE;
        switch (lastName.compareTo(o.lastName)) {
        case -1:
            result = -1;
            break;
        case 1:
            result = 1;
            break;
        case 0:
            result = (firstName.compareTo(o.firstName));
            break;
        }
        return result;
    }
    @Override
    public String toString() {
        return "[" + lastName + ", " + firstName + "]";
    }
}
```

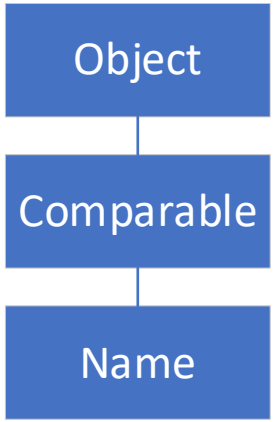
Name.java | Main.java

```java
public class Main {
    public static void main(String[] args) {
        Name n1 = new Name("Zonk", "Godot");
        Name n2 = new Name("Bogdanov", "Alexei");
        Name n3 = new Name("Ace", "Portgas D.");
        Name names[] = {n1, n2, n3};
        System.out.println("Original order:");
        for (Name n : names) {
            System.out.println(n);
        }
        java.util.Arrays.sort(names);
        System.out.println("\nAfter sorting:");
        for (Name n : names) {
            System.out.println(n);
        }

    }

}
```

```
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:

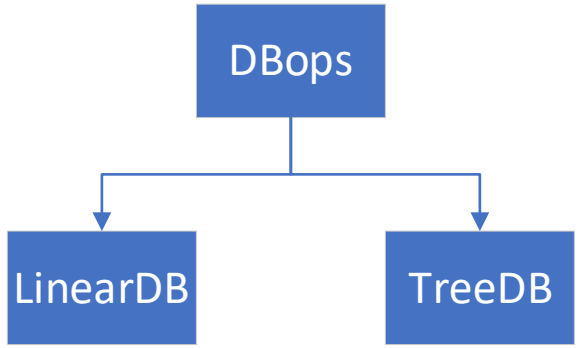
```
addActionListener(ActionListener)
removeActionListener(ActionListener)
```

`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`?