OOP with Java

24. Interfaces

Thomas Weise · 汤卫思

tweise@hfuu.edu.cn · http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Faculty of Computer Science and Technology
Institute of Applied Optimization
230601 Shushan District, Hefei, Anhui, China
Econ. & Tech. Devel. Zone, Jinxiu Dadao 99

合肥学院 南艳湖校区/南2区
计算机科学与技术系
应用优化研究所
中国 安徽省 合肥市 蜀山区 230601
经济技术开发区 锦绣大道99号

Outline

1. Introduction
2. Interfaces: Definition, Usage
3. Default Methods
4. Interfaces in Java
5. Summary

website

Introduction

- With `abstract` classes we can now define an “API” which can be implemented by subclasses in different ways

Introduction

- With `abstract` classes we can now define an "API" which can be implemented by subclasses in different ways
- In Java, we have "single inheritance", which means that each class has exactly one super-class that it extends (except for `Object`, which has none)

Introduction

- With `abstract` classes we can now define an “API” which can be implemented by subclasses in different ways
- In Java, we have “single inheritance”, which means that each class has exactly one super-class that it extends (except for `Object`, which has none)
- This means that if we want to define multiple independent APIs (in different `abstract` classes) and implement them in one class

Introduction

- With `abstract` classes we can now define an "API" which can be implemented by subclasses in different ways
- In Java, we have "single inheritance", which means that each class has exactly one super-class that it extends (except for `Object`, which has none)
- This means that if we want to define multiple independent APIs (in different `abstract` classes) and implement them in one class, we have a problem

Introduction

- With `abstract` classes we can now define an "API" which can be implemented by subclasses in different ways
- In Java, we have "single inheritance", which means that each class has exactly one super-class that it extends (except for `Object`, which has none)
- This means that if we want to define multiple independent APIs (in different `abstract` classes) and implement them in one class, we have a problem, because this is not possible

Introduction

- With `abstract` classes we can now define an "API" which can be implemented by subclasses in different ways
- In Java, we have "single inheritance", which means that each class has exactly one super-class that it extends (except for `Object`, which has none)
- This means that if we want to define multiple independent APIs (in different `abstract` classes) and implement them in one class, we have a problem, because this is not possible
- But this is actually not an uncommon case

Introduction

- With `abstract` classes we can now define an "API" which can be implemented by subclasses in different ways
- In Java, we have "single inheritance", which means that each class has exactly one super-class that it extends (except for `Object`, which has none)
- This means that if we want to define multiple independent APIs (in different `abstract` classes) and implement them in one class, we have a problem, because this is not possible
- But this is actually not an uncommon case
- For this, there exist interfaces

Interfaces

- An `interface` is a type, similar to a class

Interfaces

- An `interface` is a type, similar to a class, that can contain only constants

Interfaces

- An `interface` is a type, similar to a class, that can contain only constants, method signatures

Interfaces

- An `interface` is a type, similar to a class, that can contain only constants, method signatures, default methods

Interfaces

- An `interface` is a type, similar to a class, that can contain only constants, method signatures, default methods, static methods

Interfaces

- An `interface` is a type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types

Interfaces

- An `interface` is a type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types
- For now, let us just focus on method signatures, i.e., `abstract` methods

Interfaces

- An `interface` is a type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types
- For now, let us just focus on method signatures, i.e., `abstract` methods
- Interfaces are declared in the same way as classes, using the keyword `interface` instead of `class`

Interfaces

- An `interface` is a type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types
- For now, let us just focus on method signatures, i.e., `abstract` methods
- Interfaces are declared in the same way as classes, using the keyword `interface` instead of `class`
- An `interface` `A` with one method `void print()` is defined as

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

Interfaces

- An `interface` is a type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types
- For now, let us just focus on method signatures, i.e., `abstract` methods
- Interfaces are declared in the same way as classes, using the keyword `interface` instead of `class`
- An `interface` `A` with one method `void print()` is defined as

  ```
  interface A { void print(); }
  ```
- Like classes, interfaces can inherit from each other via `extends`, but an interface can have multiple super-interfaces

Interfaces

- An `interface` is a type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types
- For now, let us just focus on method signatures, i.e., `abstract` methods
- Interfaces are declared in the same way as classes, using the keyword `interface` instead of `class`
- An `interface` `A` with one method `void print()` is defined as `interface A { void print(); }`
- Like classes, interfaces can inherit from each other via `extends`, but an interface can have multiple super-interfaces
- An `interface` `A` can be implemented by a class `B` by declaring it as `class B implements A` and implementing (overriding) all of the interface methods

Interfaces

- An `interface` is a type, similar to a class, that can contain only constants, method signatures, default methods, static methods, and nested types
- For now, let us just focus on method signatures, i.e., `abstract` methods
- Interfaces are declared in the same way as classes, using the keyword `interface` instead of `class`
- An `interface` `A` with one method `void print()` is defined as `interface A { void print(); }`
- Like classes, interfaces can inherit from each other via `extends`, but an interface can have multiple super-interfaces
- An `interface` `A` can be implemented by a class `B` by declaring it as `class B implements A` and implementing (overriding) all of the interface methods
- A class can implement any number of interfaces

An example for an `interface`

Listing: An example for an StringFunction

```java
/** an interface for string functions */
public interface StringFunction {

  /** compute the result of the string function for a given input
   * @param in the input string
   * @return the result of the string function */
  public String compute(final String in);

  /** a static function which maps all the strings in an array and
   *  prints them
   * @param strings the strings
   * @param func the mapper function
   */
  public static void mapAndPrint(String[] strings, StringFunction func) {
    for(String string : strings) { // fast read-only iteration over array
      System.out.print(func.compute(string)); // print function result
      System.out.print('␣');                  // print space
    }
  }
}
```

An example implementation of the `interface`

Listing: An example implementation of the StringFunction

```java
/** a string function just rendering each string as upper case */
public class UpperCase implements StringFunction {
  /** convert a string to upper case */
  @Override
  public final String compute(String in) {
    return in.toUpperCase(); // convert String in to upper case: "a" -> "A"
  }

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    StringFunction.mapAndPrint(new String[] { // allocate text, apply function
        "Hello", "World!", "It's",  //$NON-NLS-1$//$NON-NLS-2$//$NON-NLS-3$
        "me,",  "your", "good",     //$NON-NLS-1$//$NON-NLS-2$//$NON-NLS-3$
        "old", "teacher."           //$NON-NLS-1$//$NON-NLS-2$
    }, new UpperCase()); // HELLO WORLD! IT'S ME, YOUR GOOD OLD TEACHER.
  }
}
```

Implementation of the interface based on HashMap

Listing: Implementation of the interface based on HashMap

```java
import java.util.HashMap;

/** A map function transforms a string according to a map. This would not be
 *  possible using an abstract base class, since then we could not extend HashMap */
public class MapFunction extends HashMap<String, String> implements StringFunction {
  /** the serial version uid, do not worry about this, just ignore it  */
  private static final long serialVersionUID = 1L;

  /** If a mapping is defined for the string in, return the mapping.
   * Otherwise, return the string in.*/
  @Override
  public final String compute(String in) { // Obtain the mapping for "in" stored in
    String replacement = this.get(in);     // this hash map. If there is one stored,
    return (replacement != null) ? replacement : in; //return it, otherwise return "in"
  }

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    MapFunction map = new MapFunction(); // create the map function
    map.put("teacher.", "Prof.␣Weise."); //$NON-NLS-1$//$NON-NLS-2$
    map.put("me,",      "your␣teacher,"); //$NON-NLS-1$//$NON-NLS-2$

    StringFunction.mapAndPrint(new String[] {//
        "Hello", "World!", "It's",  //$NON-NLS-1$//$NON-NLS-2$//$NON-NLS-3$
        "me,",  "your", "good",     //$NON-NLS-1$//$NON-NLS-2$//$NON-NLS-3$
        "old", "teacher."           //$NON-NLS-1$//$NON-NLS-2$
    }, map); // Hello World! It's your teacher, your good old Prof. Weise.
  }
}
```

An example for a generic `interface`

Listing: A generic interface allowing to add elements

```java
/** a generic interface allowing to add elements.
 * @param <T> the type of elements to add  */
public interface Addable<T> {

  /** add a value */
  public void addAtEnd(final T value);

}
```

An class implementing two interfaces

Listing: An class implementing two interfaces

```java
import java.util.ArrayList;

/** a class implementing two interfaces, StringFunction and Addable<StringFunction>, on top of class ArrayList */
public class ConcatenatedFunction extends ArrayList<StringFunction> implements StringFunction, Addable<StringFunction> {
  /** the serial version uid, do not worry about this, just ignore it  */
  private static final long serialVersionUID = 1L;

  /** convert a string by applying all functions one by one */
  @Override
  public final String compute(String in) {
    String current = in;                       // start at the current string c
    for(StringFunction function : this) {  // for each function f in this ArrayList
      current = function.compute(current); // set c ⟵ f(c)
    }
    return current;                            // return the resulting string
  }

  /** implement addAtEnd from interface Addable */
  @Override
  public void addAtEnd(StringFunction value) {
    this.add(value); // add value to the list of functions to carry out
  }

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    MapFunction map = new MapFunction();  // create a MapFunction
    map.put("teacher.", "Prof.␣Weise.");  // replace "teacher." with "Prof. Weise"  //$NON-NLS-1$//$NON-NLS-2$
    map.put("me,",      "your␣teacher,"); // replace "me,"      with "your teacher" //$NON-NLS-1$//$NON-NLS-2$

    ConcatenatedFunction func = new ConcatenatedFunction(); // create a concatenated function
    func.addAtEnd(map);                                     // tell it to first perform the map function
    func.addAtEnd(new UpperCase());                         // and then to convert the result to upper case

    StringFunction.mapAndPrint(new String[] { // allocate an array containing 8 strings and appy the function to them
        "Hello", "World!", "It's",  //$NON-NLS-1$//$NON-NLS-2$//$NON-NLS-3$
        "me,",  "your", "good",     //$NON-NLS-1$//$NON-NLS-2$//$NON-NLS-3$
        "old", "teacher."           //$NON-NLS-1$//$NON-NLS-2$
    }, func); // HELLO WORLD! IT'S YOUR TEACHER, YOUR GOOD OLD PROF. WEISE.
  }
}
```

An example for an `interface` subclassing another one

Listing: An example for an `interface` subclassing another one

```java
/** An interface extending StringFunction by an additional function */
public interface InvertibleStringFunction extends StringFunction {

  /** get a string function which is the inverse of this one, i.e.,
   * if this function maps "A" to "B", the resulting function should
   * map "B" to "A" */
  public StringFunction invert();
}
```

An class implementing that interface

Listing: An class implementing that interface

```
import java.util.HashMap;

/** A map function transforms a string according to a map, able to create an inverse function.
 * This class implements InvertibleStringFunction and thus also implements its super-interface
 * StringFunction. */
public class ReversibleMapFunction extends HashMap<String, String> implements InvertibleStringFunction {
  /** the serial version uid, do not worry about this, just ignore it  */
  private static final long serialVersionUID = 1L;

  /** If a mapping is defined for the string in, return the mapping.
   * Otherwise, return the string in.*/
  @Override
  public final String compute(String in) { // Obtain the mapping for "in" stored in
    String replacement = this.get(in);     // this hash map. If there is one stored,
    return (replacement != null) ? replacement : in; //return it, otherwise return "in"
  }

  /** return the inverse function (notice the more specific return type) */
  @Override
  public ReversibleMapFunction invert() {                          // create an inverse function
    ReversibleMapFunction inverse = new ReversibleMapFunction(); // allocate the object
    for(Entry<String,String> entry : this.entrySet()) {          // iterate over all key-value entries in the map
      inverse.put(entry.getValue(), entry.getKey());             // for each A -> B mapping in this function
    }                                                            //  add a B -> A mapping in the new function
    return inverse;                                              // and return the result
  }

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    ReversibleMapFunction map = new ReversibleMapFunction(); // create the reversible map function
    map.put("teacher.", "Prof.␣Weise.");  // replace "teacher." with "Prof. Weise"  //$NON-NLS-1$//$NON-NLS-2$
    map.put("me,",      "your␣teacher,"); // replace "me,"      with "your teacher" //$NON-NLS-1$//$NON-NLS-2$

    ConcatenatedFunction func = new ConcatenatedFunction(); // create a concatenated function
    func.add(map);          // add function
    func.add(map.invert()); // add inverse function

    StringFunction.mapAndPrint(new String[] {//
        "Hello", "World!", "It's",  //$NON-NLS-1$//$NON-NLS-2$//$NON-NLS-3$
        "me,",  "your", "good",     //$NON-NLS-1$//$NON-NLS-2$//$NON-NLS-3$
        "old", "teacher."           //$NON-NLS-1$//$NON-NLS-2$
    }, func); // Hello World! It's me, your good old teacher.
  }
}
```

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation
  - they can have `static` methods and `public static final` variables

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation
  - they can have `static` methods and `public static final` variables
  - they can extend another class / interface, respectively

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation
  - they can have `static` methods and `public static final` variables
  - they can extend another class / interface, respectively
  - they can have arbitrarily many subclasses / sub-interfaces extending them, respectively

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation
  - they can have `static` methods and `public static final` variables
  - they can extend another class / interface, respectively
  - they can have arbitrarily many subclasses / sub-interfaces extending them, respectively
  - they can be generic

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation
  - they can have `static` methods and `public static final` variables
  - they can extend another class / interface, respectively
  - they can have arbitrarily many subclasses / sub-interfaces extending them, respectively
  - they can be generic
  - a variable of a `class` type can be assigned as value any of the type's sub-classes and a variable of an `interface` type can be assigned as value any of the type's sub-interfaces

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation
  - they can have `static` methods and `public static final` variables
  - they can extend another class / interface, respectively
  - they can have arbitrarily many subclasses / sub-interfaces extending them, respectively
  - they can be generic
  - a variable of a `class` type can be assigned as value any of the type's sub-classes and a variable of an `interface` type can be assigned as value any of the type's sub-interfaces
- But they also differ in many ways

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation
  - they can have `static` methods and `public static final` variables
  - they can extend another class / interface, respectively
  - they can have arbitrarily many subclasses / sub-interfaces extending them, respectively
  - they can be generic
  - a variable of a `class` type can be assigned as value any of the type's sub-classes and a variable of an `interface` type can be assigned as value any of the type's sub-interfaces
- But they also differ in many ways:
  - classes can have instance variables, interfaces cannot

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation
  - they can have `static` methods and `public static final` variables
  - they can extend another class / interface, respectively
  - they can have arbitrarily many subclasses / sub-interfaces extending them, respectively
  - they can be generic
  - a variable of a `class` type can be assigned as value any of the type's sub-classes and a variable of an `interface` type can be assigned as value any of the type's sub-interfaces
- But they also differ in many ways:
  - classes can have instance variables, interfaces cannot
  - classes can have `private`, package-private, or `protected` methods, while interfaces can have only `public` methods (regardless whether `static` or not)

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation
  - they can have `static` methods and `public static final` variables
  - they can extend another class / interface, respectively
  - they can have arbitrarily many subclasses / sub-interfaces extending them, respectively
  - they can be generic
  - a variable of a `class` type can be assigned as value any of the type's sub-classes and a variable of an `interface` type can be assigned as value any of the type's sub-interfaces
- But they also differ in many ways:
  - classes can have instance variables, interfaces cannot
  - classes can have `private`, package-private, or `protected` methods, while interfaces can have only `public` methods (regardless whether `static` or not)
  - classes must extend exactly one superclass, interfaces can extend arbitrarily many super-interfaces (or none), but no classes

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation
  - they can have `static` methods and `public static final` variables
  - they can extend another class / interface, respectively
  - they can have arbitrarily many subclasses / sub-interfaces extending them, respectively
  - they can be generic
  - a variable of a `class` type can be assigned as value any of the type's sub-classes and a variable of an `interface` type can be assigned as value any of the type's sub-interfaces
- But they also differ in many ways:
  - classes can have instance variables, interfaces cannot
  - classes can have `private`, package-private, or `protected` methods, while interfaces can have only `public` methods (regardless whether `static` or not)
  - classes must extend exactly one superclass, interfaces can extend arbitrarily many super-interfaces (or none), but no classes
  - classes can implement arbitrarily many interfaces, while interfaces cannot implement anything

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation
  - they can have `static` methods and `public static final` variables
  - they can extend another class / interface, respectively
  - they can have arbitrarily many subclasses / sub-interfaces extending them, respectively
  - they can be generic
  - a variable of a `class` type can be assigned as value any of the type's sub-classes and a variable of an `interface` type can be assigned as value any of the type's sub-interfaces
- But they also differ in many ways:
  - classes can have instance variables, interfaces cannot
  - classes can have `private`, package-private, or `protected` methods, while interfaces can have only `public` methods (regardless whether `static` or not)
  - classes must extend exactly one superclass, interfaces can extend arbitrarily many super-interfaces (or none), but no classes
  - classes can implement arbitrarily many interfaces, while interfaces cannot implement anything
  - classes can implement methods, while interfaces cannot

Interfaces vs. Classes

- So we have seen quite a few examples for using interfaces.
- Interfaces and `abstract` classes have a lot in common:
  - they can have method specifications (signatures) without implementation
  - they can have `static` methods and `public static final` variables
  - they can extend another class / interface, respectively
  - they can have arbitrarily many subclasses / sub-interfaces extending them, respectively
  - they can be generic
  - a variable of a `class` type can be assigned as value any of the type's sub-classes and a variable of an `interface` type can be assigned as value any of the type's sub-interfaces
- But they also differ in many ways:
  - classes can have instance variables, interfaces cannot
  - classes can have `private`, package-private, or `protected` methods, while interfaces can have only `public` methods (regardless whether `static` or not)
  - classes must extend exactly one superclass, interfaces can extend arbitrarily many super-interfaces (or none), but no classes
  - classes can implement arbitrarily many interfaces, while interfaces cannot implement anything
  - classes can implement methods, while interfaces cannot … well, actually …

Default Methods

- Since Java 8, an `interface` can not just specify a method's signature, but also a default implementation

Default Methods

- Since Java 8, an `interface` can not just specify a method's signature, but also a default implementation
- This is mainly intended for situations where an interface provides some very basic functionality and higher-level functions on top of that which could be implemented only using the basic functions

Default Methods

- Since Java 8, an `interface` can not just specify a method's signature, but also a default implementation
- This is mainly intended for situations where an interface provides some very basic functionality and higher-level functions on top of that which could be implemented only using the basic functions
- This higher-level functionality then goes into a default method and in an implementing class, we just need to override the methods with the basic functionality

Default Methods

- Since Java 8, an `interface` can not just specify a method's signature, but also a default implementation
- This is mainly intended for situations where an interface provides some very basic functionality and higher-level functions on top of that which could be implemented only using the basic functions
- This higher-level functionality then goes into a default method and in an implementing class, we just need to override the methods with the basic functionality
- Default implementations of methods are declared in the form `public default ...rest-of-signature` and then must have a function body

Default Methods

- Since Java 8, an `interface` can not just specify a method's signature, but also a default implementation
- This is mainly intended for situations where an interface provides some very basic functionality and higher-level functions on top of that which could be implemented only using the basic functions
- This higher-level functionality then goes into a default method and in an implementing class, we just need to override the methods with the basic functionality
- Default implementations of methods are declared in the form `public default ...rest-of-signature` and then must have a function body
- They can still be implemented by classes implementing the interface, but do not need to be implemented

An interface with a `default` method

Listing: An interface with a default method

```java
/** an interface allowing us to read some text */
public interface TextSource {
  /** read a single character, returning -1 if no more text is available */
  public int readChar();

  /** read a full line of text, returns a non-empty string or, if there is no
   *  more text, {@code null} */
  public default String readLine() {
    int chr;

    String line = "";//$NON-NLS-1$

    for (;;) {                         // repeat until we got a line
      switch (chr = this.readChar()) { // read the next character (into variable chr)
        case '\n':                     // newline
        case '\r': {                   // (used in windows \r\n)
          line = line.trim();          // remove leading and trailing spaces
          if (line.length() <= 0) {    // if line is empty,
            continue;                  // then let's try again (deal with \r\n)
          }                            // otherwise...
        }                              // no return/break: fall-through to handling -1

        //$FALL-THROUGH$               // falling through from above
        case -1: {                     // -1 means we have reached the end of the text
          return (line.length() <= 0) ? null : line; // if line is empty, return null
        }                              // otherwise return line. null only happens at end

        default: {                     // if we get here, there was neither \r, \n\ nor -1
          line += ((char) chr);        // so we add the character we read to the line
        }
      }
    }
  }
}
```

An implementation of this interface

Listing: An implementation of this interface

```java
/** a string-based text source */
public class StringTextSource implements TextSource {
  /** the string */
  String string;
  /** the current index */
  int index;

  /** create the text source */
  StringTextSource(final String _string) {
    this.string = _string; // store the string, index will be 0
  }

  /** read a character */
  @Override
  public int readChar() {
    if (this.index < this.string.length()) {   // if we did not reach end of string,
      return this.string.charAt(this.index++); // return character at index, then increase index
    }
    return -1;                                 // we have reached end: return -1
  }

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {

    TextSource source = new StringTextSource(// create text source with 3 non-white-space lines of text as 1 string
              "Hello␣World!\n␣␣It␣is␣me!\n␣\r\nYour␣friendly␣teacher!"); //$NON-NLS-1$

    String current;
    while ((current = source.readLine()) != null) { // as long as we did not yet reach the end
      System.out.println(current);                  // print the current string
    }
  }
}
```

Interfaces for APIs

- Interfaces are a great way to specify an API

Interfaces for APIs

- Interfaces are a great way to specify an API
- The users of such an API just see the interfaces and do not need to care about the implementation

Interfaces for APIs

- Interfaces are a great way to specify an API
- The users of such an API just see the interfaces and do not need to care about the implementation
- They will never feel tempted to look into code, instance variables, etc, since there are none

Interfaces for APIs

- Interfaces are a great way to specify an API
- The users of such an API just see the interfaces and do not need to care about the implementation
- They will never feel tempted to look into code, instance variables, etc, since there are none
- The implementors of such an API can use whatever base classes and external libraries they like

Interfaces for APIs

- Interfaces are a great way to specify an API
- The users of such an API just see the interfaces and do not need to care about the implementation
- They will never feel tempted to look into code, instance variables, etc, since there are none
- The implementors of such an API can use whatever base classes and external libraries they like
- They only need to make sure to fulfill the interface specification and don't need to worry about anything else

Interfaces for APIs

- Interfaces are a great way to specify an API
- The users of such an API just see the interfaces and do not need to care about the implementation
- They will never feel tempted to look into code, instance variables, etc, since there are none
- The implementors of such an API can use whatever base classes and external libraries they like
- They only need to make sure to fulfill the interface specification and don't need to worry about anything else
- APIs specified via interfaces can be implemented several times, using different approaches and classes, by different people

Interfaces for APIs

- Interfaces are a great way to specify an API
- The users of such an API just see the interfaces and do not need to care about the implementation
- They will never feel tempted to look into code, instance variables, etc, since there are none
- The implementors of such an API can use whatever base classes and external libraries they like
- They only need to make sure to fulfill the interface specification and don't need to worry about anything else
- APIs specified via interfaces can be implemented several times, using different approaches and classes, by different people
- A user can switch from one implementation to another one without problems

Interfaces for APIs

- Interfaces are a great way to specify an API
- The users of such an API just see the interfaces and do not need to care about the implementation
- They will never feel tempted to look into code, instance variables, etc, since there are none
- The implementors of such an API can use whatever base classes and external libraries they like
- They only need to make sure to fulfill the interface specification and don't need to worry about anything else
- APIs specified via interfaces can be implemented several times, using different approaches and classes, by different people
- A user can switch from one implementation to another one without problems
- Java's standard classes massively make use of interfaces

Example: Interfaces and Collections

- The Java collection framework is actually specified as set of interfaces

Example: Interfaces and Collections

- The Java collection framework is actually specified as set of interfaces:
    - `java.util.List` is an interface describing the list functionality and is implemented (indirectly) by, e.g., `java.util.ArrayList`

Example: Interfaces and Collections

- The Java collection framework is actually specified as set of interfaces:
  - `java.util.List` is an interface describing the list functionality and is implemented (indirectly) by, e.g., `java.util.ArrayList`
  - `java.util.Set` is the same for sets and `java.util.Map` for maps

Example: Interfaces and Collections

- The Java collection framework is actually specified as set of interfaces:
  - `java.util.List` is an interface describing the list functionality and is implemented (indirectly) by, e.g., `java.util.ArrayList`
  - `java.util.Set` is the same for sets and `java.util.Map` for maps
  - `java.util.Collection` is a super-interface of `java.util.Set` and `java.util.List` defining just a collection of objects

Example: Interfaces and Collections

- The Java collection framework is actually specified as set of interfaces:
  - `java.util.List` is an interface describing the list functionality and is implemented (indirectly) by, e.g., `java.util.ArrayList`
  - `java.util.Set` is the same for sets and `java.util.Map` for maps
  - `java.util.Collection` is a super-interface of `java.util.Set` and `java.util.List` defining just a collection of objects
  - `java.util.Iterator` represents a one-time iteration over a sequence

Example: Interfaces and Collections

- The Java collection framework is actually specified as set of interfaces:
  - `java.util.List` is an interface describing the list functionality and is implemented (indirectly) by, e.g., `java.util.ArrayList`
  - `java.util.Set` is the same for sets and `java.util.Map` for maps
  - `java.util.Collection` is a super-interface of `java.util.Set` and `java.util.List` defining just a collection of objects
  - `java.util.Iterator` represents a one-time iteration over a sequence
  - `java.lang.Iterable` can be implemented by anything which can be iterated over (which can create an instance of `java.util.Iterator` on its elements)

Example: Interfaces and Collections

- The Java collection framework is actually specified as set of interfaces:
  - `java.util.List` is an interface describing the list functionality and is implemented (indirectly) by, e.g., `java.util.ArrayList`
  - `java.util.Set` is the same for sets and `java.util.Map` for maps
  - `java.util.Collection` is a super-interface of `java.util.Set` and `java.util.List` defining just a collection of objects
  - `java.util.Iterator` represents a one-time iteration over a sequence
  - `java.lang.Iterable` can be implemented by anything which can be iterated over (which can create an instance of `java.util.Iterator` on its elements), it is a super-interface of `java.util.Collection`
  - Java supports `java.lang.Iterable`: `for(T x: ...)` works not just with arrays, but also iterates over an iterator provided by `Iterable`

Using Collections and the interface `java.util.Iterator`

Listing: Using Collections and the interface java.util.Iterator

```java
import java.util.ArrayList;
import java.util.Iterator;

/** we use ArrayList and an Iterator on its elements */
public class IteratorTest {

  /** The main routine
   * @param args
   *          we ignore this parameter for now */
  public static final void main(String[] args) {
    ArrayList<String> list = new ArrayList<>();

    list.add("Hello"); list.add("World!");   //$NON-NLS-1$//$NON-NLS-2$
    list.add("It's");  list.add("me");       //$NON-NLS-1$//$NON-NLS-2$
    list.add("your");  list.add("teacher."); //$NON-NLS-1$//$NON-NLS-2$

 // create the iterator: this method is inherited from Iterable
    Iterator<String> iterator = list.iterator();

    while(iterator.hasNext()) { // hasNext returns true = there are more elements
      System.out.print(iterator.next()); // next returns the next element
      System.out.print('␣');
    } // Hello World! It's me your teacher.
  }
}
```

Using the Simplified Syntax for Iterations

Listing: Using the Simplified Syntax for Iterations via `java.lang.Iterable`

```java
import java.util.ArrayList;
import java.util.Iterator;

/** we use ArrayList and test the syntactical sugar; equivalent to IteatorTest */
public class IterableTest {

  /** The main routine
   * @param args
   *         we ignore this parameter for now */
  public static final void main(String[] args) {
    ArrayList<String> list = new ArrayList<>();

    list.add("Hello"); list.add("World!");   //$NON-NLS-1$//$NON-NLS-2$
    list.add("It's");  list.add("me");       //$NON-NLS-1$//$NON-NLS-2$
    list.add("your");  list.add("teacher."); //$NON-NLS-1$//$NON-NLS-2$

    // this creates an iterator by using list.iterator, and then iterates
    for(String string : list) { // over the list, storing the elements in string
      System.out.print(string); // one by one, and here we print them
      System.out.print('␣');
    } // Hello World! It's me your teacher.
  }
}
```

Summary

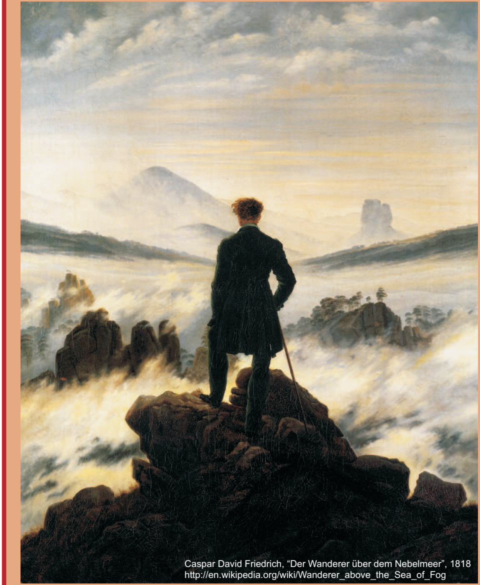
- We have learned about interfaces
- Interfaces allow us to specify API contracts in form of method signatures which must be implemented by an implementor and can be used by a user (both programmers, obviously)
- Interfaces have many similarities with classes, but also several differences
- Interfaces cannot be instantiated directly (well, actually … but let's leave this for later) and instead need to be implemented
- Interfaces can inherit from multiple other interfaces
- Classes can implement multiple interfaces
- Since Java 8, interfaces can have default method implementations
- Java massively uses interfaces in its API, for example in the Collections API

谢谢

Thank you

Thomas Weise [汤卫思]
tweise@hfuu.edu.cn
http://iao.hfuu.edu.cn

Hefei University, South Campus 2
Institute of Applied Optimization
Shushan District, Hefei, Anhui,
China



Caspar David Friedrich, “Der Wanderer über dem Nebelmeer”, 1818
http://en.wikipedia.org/wiki/Wanderer_above_the_Sea_of_Fog