Java Programming Module Interfaces, Collections and Generics

DHBW Duale Hochschule Baden-Württemberg Mannheim

Prof. Dr. Holger D. Hofmann

Polymorphism and Interfaces

Recap

```
package polymorphism;

public class clsMain {

public static void main(String[] args) {
     clsPerson p = new clsPerson();
    clsStudent s = new clsStudent();

    clsPerson refPerson;
    clsStudent refStudent;
    ICallable refInterface;

    refPerson = p;
    refPerson = s; //a student is also a person

    refStudent = s;
    refStudent = p; //does not work!!!

    refInterface = p;//polymorphism via interface!
    refInterface = s;//polymorphism via interface!
}
}
```

Inheritance in Java

- Inheritance builds an is-a relationship (a student is-a person)
- Single Class inheritance results in reusing
  - the class‘s Signature and
  - its Implementation

  of the base class

```
public class Student extends Person
```

- Multiple Interace implementation results in reusing
  - the interfaces‘ signature

```
public class Student implements IPerson
```

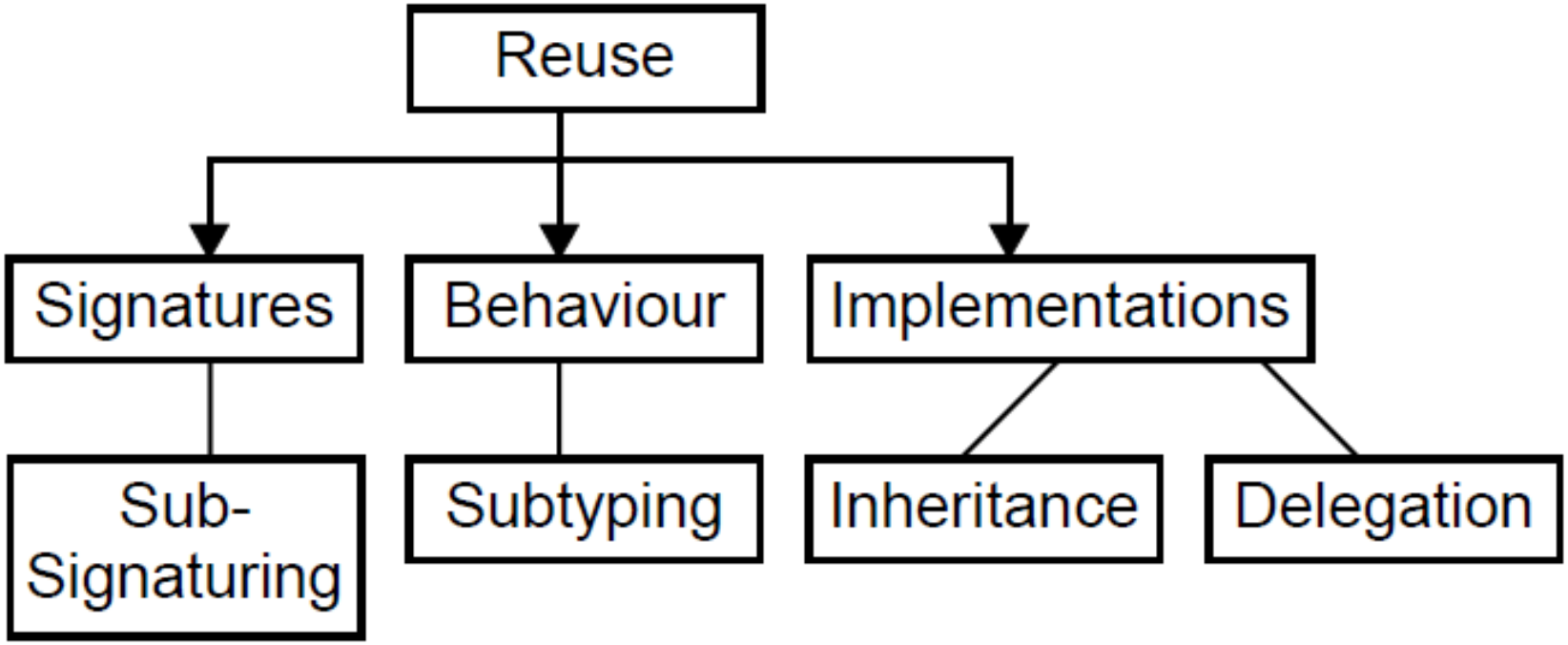
- Multiple Interface inheritance results in reusing
  - the interface‘s signature

```
interface IStudent extends IPerson
```

Reuse

- Reuse means not to re-invent the wheel
- Abstractions as well as implementations can be reused
- OO Reuse: Inheritance or Delegation
- Types of reuse
  - Black-box reuse
  - White-box reuse
  - Glass-box reuse

Taxonomy of Reuse



Hofmann, H.D.; Stynes, J. Implementation Reuse and Inheritance in Distributed Component Systems. COMPSAC, Wien, 1998.

Interface Inheritance

- Interfaces can be derived from other interfaces
- Caution: Java supports single implementation inheritance, but multiple interface inheritance

```
public interface IBasic1 {
int getInt();
}

public interface IBasic2 {
float getFloat();
}

public interface IDerived extends
IBasic1, IBasic2{
double getDouble();
}
```

```
public class clsTest implements
IDerived {

public int getInt()
{return 4711;}

public float getFloat()
{return 3/8;}

public double getDouble()
{return 471111;}

}
```

The Interface Comparable

- Provides Comparison for Object independantly of the class hierarchy

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

- Compareto returns
  - <0 if current Elements is "smaller than" o
  - >0 if current Element is "greater than" o
  - 0 if both are equal

Implementing Multiple Interfaces

- One Class may implement multiple Interfaces
- Comparable to Effects of Multiple Implementation Inheritance, but w/o side effects
- Integrating an Object into different inheritance hierarchies via implementing multiple interfaces

Example: Implementing Multiple Interfaces

```
001 /* Auto3.java */
002
003 public class Auto3
004 implements Size, Comparable
005 {
006   public String name;
007   public int    erstzulassung;
008   public int    leistung;
009   public int    laenge;
010   public int    hoehe;
011   public int    breite;
012
013   public int laenge()
014   {
015     return this.laenge;
016   }
017
018   public int hoehe()
019   {
020     return this.hoehe;
021   }
022
023   public int breite()
024   {
025     return this.breite;
026   }
027
028   public int compareTo(Object o)
029   {
030     int ret = 0;
031     if (leistung < ((Auto3)o).leistung) {
032       ret = -1;
033     } else if (leistung > ((Auto3)o).leistung) {
034       ret = 1;
035     }
036     return ret;
037   }
038 }
```

Interface Applications

- Constants in Interfaces

```
001 interface IConstants {
002 public static final int MONDAY = 0;
003 public static final int TUESDAY = 1;
004 // ...
005 }
006
007 public class clsMyClass
008    implements IConstants {
009 //...
010  if (nWeekday == TUESDAY) {
011 //... }
```

- Actually not intended by Java developers
- -> use of static imports (unqualified access w/o inheritance)

```
001 /* Listing0912.java */
002
003 import static java.lang.Math.*;
004
005 public class Listing0912
006 {
007   public static void main(String[] args)
008   {
009     System.out.println(sqrt(2));
010     System.out.println(sin(PI));
011   }
012 }
```

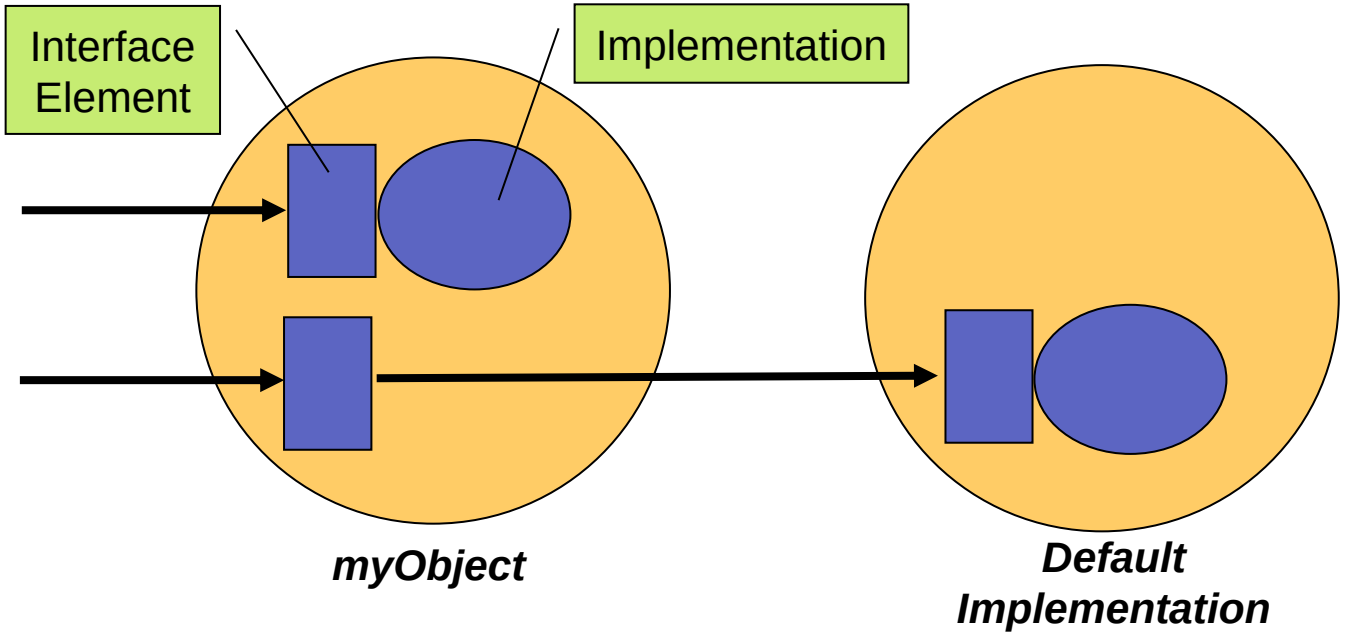
Interfaces: Default Implementation

- In contrast to abstract classes, interfaces do not provide any implementation
- For interfaces that may be implemented quite often, a default implementation can be provided as abstract classes

```
001 /* SimpleTreeNode.java */
002
003 public interface SimpleTreeNode
004 {
005   public void addChild(SimpleTreeNode child);
006   public int getChildCnt();
007   public SimpleTreeNode getChild(int pos);
008 }
```

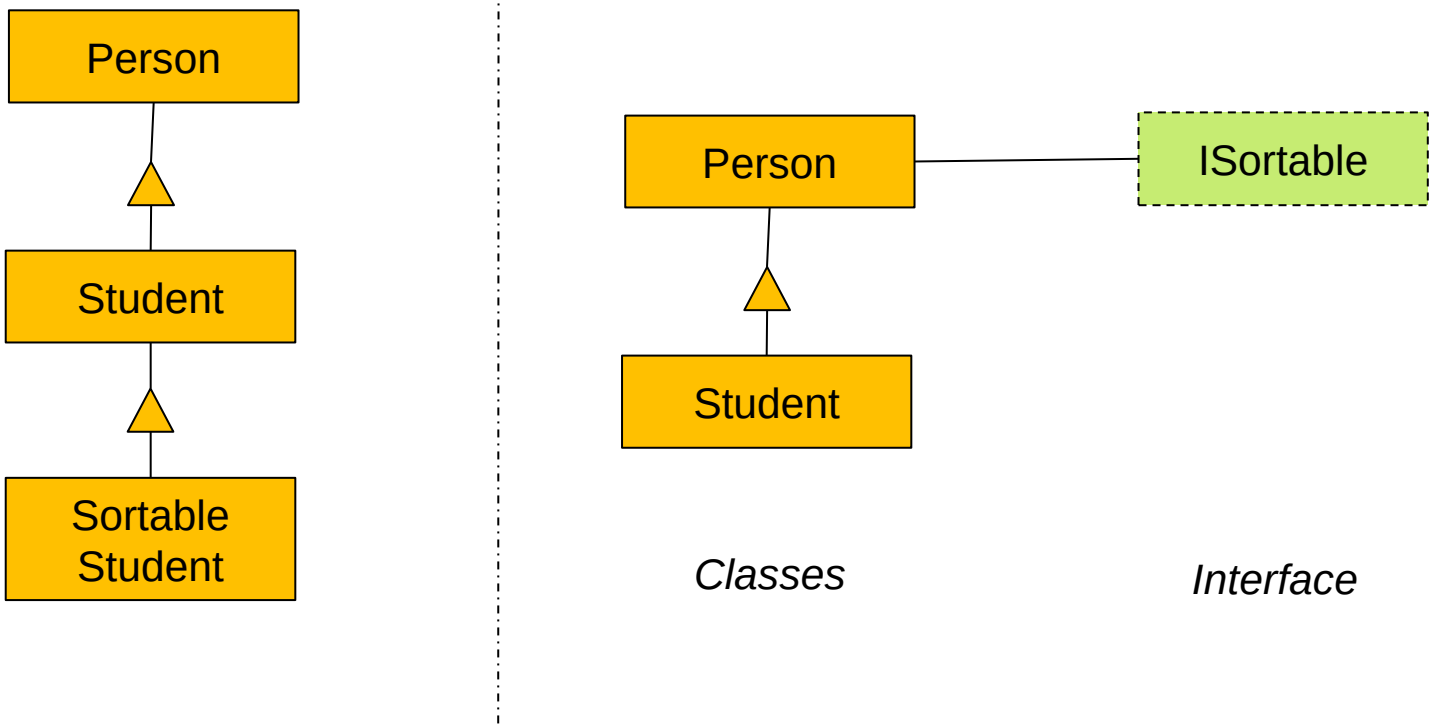
Reuse of Default Implementation: Delegation

- Classes implementing an Intefaces may
  - not inherit from the Default Implementation
  - not completely implement all methods (DRY – don't repeat yourself!)
- Solution: Delgation to Default Implementation



Class vs. Interface Inheritance

- Interface inheritance decouples an object's signature from the (interface) inheritance hierarchy



Java >7: Default Implementations

```
01 public interface TimeClient {
02    void setTime(int hour, int minute, int second);
03    void setDate(int day, int month, int year);
04
05   static ZoneId getZoneId (String zoneString) {
06        try {
07            return ZoneId.of(zoneString);
08        } catch (DateTimeException e) {
09            System.err.println("Invalid time zone: " + zoneString +
10                "; using default time zone instead.");
11            return ZoneId.systemDefault();
12        }
13    }
14
15    default ZonedDateTime getZonedDateTime(String zoneString) {
16        return ZonedDateTime.of(getLocalDateTime(), getZoneId(zoneString));
17    }
18 }
```

Example Default Implementation

```
public interface IPerson {
        default void getUp() { System.out.println("Get Up at 09:00");  }
}

public interface IEmployee {
        default void getUp() { System.out.println("Get up at 07:00");  }
}

Public interface IDHBWStudent extends IPerson, IEmployee {

}

public class clsDHBWStudent implements IDHBWStudent {

}

public class clsMain {
        public static void main(String[] args) {
                IDHBWStudent myStudent = new clsDHBWStudent();
                myStudent.getUp();
        }
}
```

```
Exception in thread "main" java.lang.Error: Unresolved compilation problem:
        Duplicate default methods named getUp with the parameters () and () are inherited from the types IEmployee and IPerson
        at clsDHBWStudent.<init>(clsDHBWStudent.java:2)
        at clsMain.main(clsMain.java:5)
```

Duplicate Default Methodes: Solution

- Conflicts in Interface

```
public interface IDHBW_Student extends IPerson, IStudent{
        default void getUp() { IPerson.super.getUp(); }
}
```

- Conflicts in implementing class

```
public class clsDHBW_Student2 implements IStudent, IPerson{
        public void getUp() { IPerson.super.getUp(); }
}
```

Collections

- Collections are classes being able to store and manage other objects
- The implement data structures such as Lists, Trees, Queues, and Stacks
- Idea (see also http://download.oracle.com/javase/6/docs/technotes/guides/collections/overview.html):
  - Handle different objects similarly
  - Have standardized Iterators
  - Have good implementations of, e.g., sorting algorithms

Collection Classes JDK <1.2

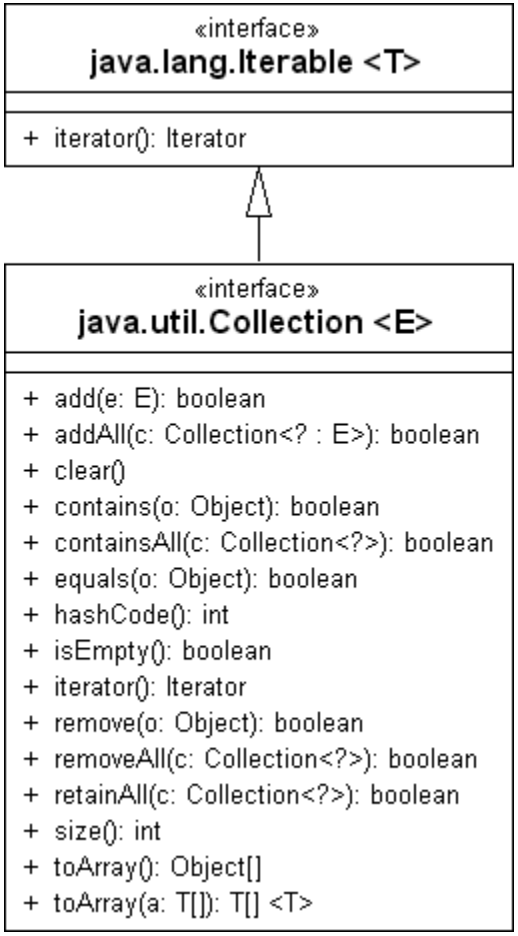
- classes Vector (array of Object objects), Stack (LIFO Vector), Dictionary (abstract class), Hashtable (implementation of Dictionary, BitSet (set of Integers)
- Synchronized, but slow
- Collections (JDK <1.2) store object of class Object -> casting and type checking required

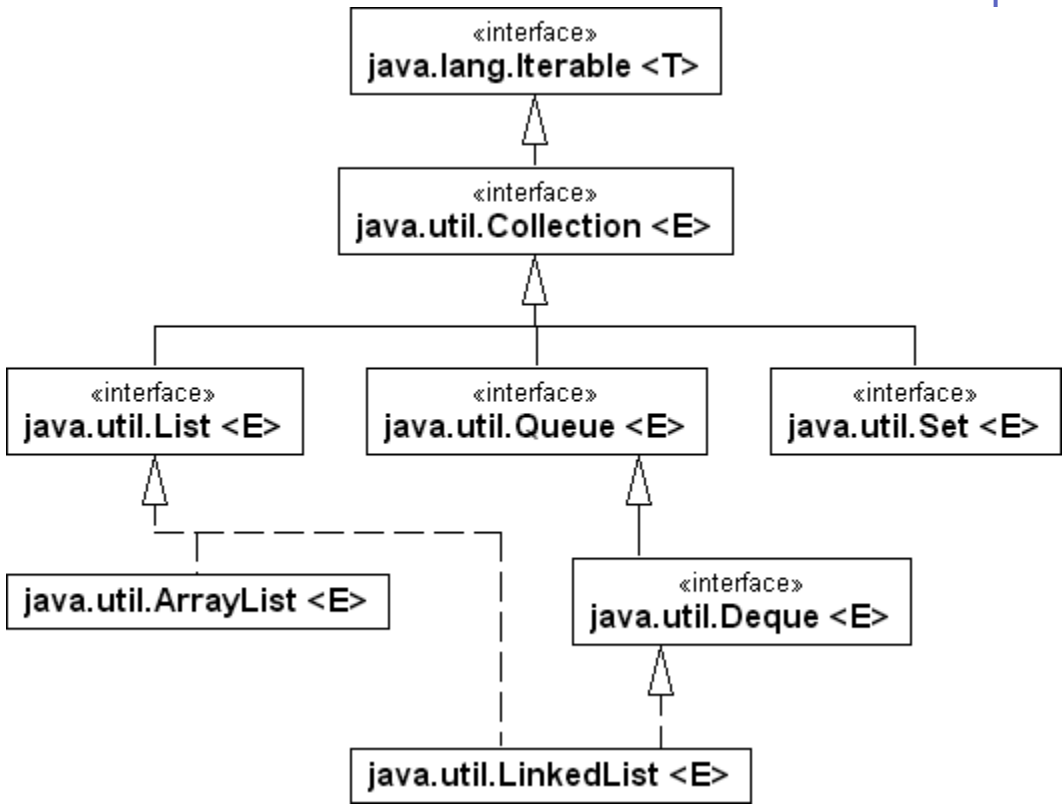
```
001 /* Listing1401.java, javabuch.de */
002
003 import java.util.*;
004
005 public class Listing1401
006 {
007   public static void main(String[] args)
008   {
009     Vector v = new Vector();
010
011     v.addElement("eins");
012     v.addElement("drei");
013     v.insertElementAt("zwei",1);
014     for (Enumeration el=v.elements(); el.hasMoreElements(); ) {
015       System.out.println((String)el.nextElement());
016     }
017   }
018 }
```

Collections (JDK since 1.2)

- All collection classes are unsynchronized
- Three different collection types (interfaces):
  - List (arbitraty objects, random or sequential access)
  - Set (set of unique values, set operations)
  - Map (object1 -> object2)

Collections (JDK since 1.2)





Source: Java ist auch eine Insel, 8. Aufl.

Generics

JDK >= 5.0

- Generics allow to abstract from concrete types, but to check for correct types
- Mainly used for Collections (list, array, set, …)
- Generics allow to narrow the type compatibility without specific implementations
- Compiler does not have to convert types
  - example: List<Integer>

Interface for List class:

```
public interface List <E>{
   void add(E x);
   Iterator<E> iterator();
}
```

Generic Collection Classes

- The definition

```
Vector v = new Vector();
```

- does not define the type to be stored in the Vector object
- This can be done using the following notation

```
Vector<Integer> v = new Vector<Integer>(); //or

Hashtable<String, Integer> h = new Hashtable<String,
  Integer>();
```

- There are three other types for parameter type definition ("bounded Wildcards"):
  Be C a class and I1 and I2 interfaces
  - <T extends C>
  - <T extends I1 & I2>
  - <T extends C & I1>
  - <T extends C & I1 & I2>

  *(see "Java ist auch eine Insel", chapter **10.5**)*

Implementing a Generic Class

```
011 public class MiniListe<E>
012 implements Iterable<E>
...
031   public void addElement(E element)
...
051   public E elementAt(int pos)
...
087   public static void main(String[] args)
088   {
089     //Untyped Usage
090     MiniListe l1 = new MiniListe(10);
091     l1.addElement(3.14);
092     l1.addElement("world");
093     for (Object o : l1) {  //Iterator
094       System.out.println(o);
095     }
096     //Ganzzahlige Typisierung
097     System.out.println("---");
098     MiniListe<Integer> l2 = new MiniListe<Integer>(5);
099     l2.addElement(3);
...
```

Code taken from MiniListe.java

Generics and Interfaces

Source: http://docs.oracle.com/javase/tutorial/java/generics/types.html

```
public interface Pair<K, V> {
    public K getKey();
    public V getValue();
}

public class OrderedPair<K, V> implements Pair<K, V> {

    private K key;
    private V value;

    public OrderedPair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey()   { return key; }
    public V getValue() { return value; }
}
```