Interfaces

1. Overview

The learning objectives of this laboratory session are:

- Understand the notion of an interface and how it differs from inheritance
- Acquire hands-on experience with existing and user-defined interfaces

1.1. What Is an Interface?

An interface is a list of methods that must be defined by any class which implements that interface. It may also define constants (public static final).

Similar to abstract class. An interface is similar to a class without instance and static variables (static final constants are allowed), and without method bodies. This is essentially what a completely abstract class does, but abstract classes do allow static method definitions, and interfaces don't.

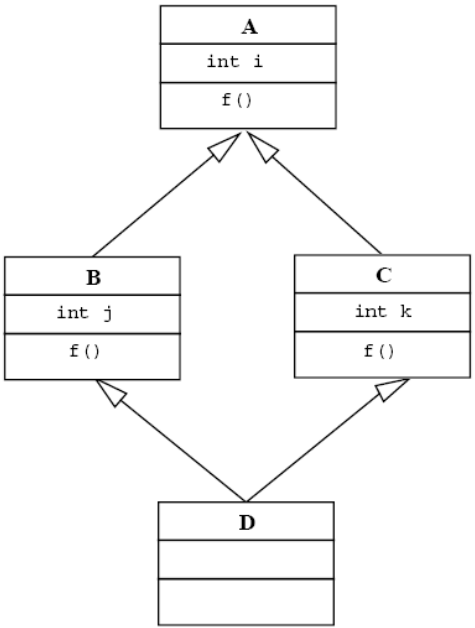
Contractual obligation. When a class specifies that it implements an interface, it must define all methods of that interface. A class can implement many different interfaces. If a class doesn't define all methods of the interfaces it agreed to define (by the *implements* clause), the compiler gives an error message, which typically says something like "This class must be declared abstract". (An abstract class is one that doesn't implement all methods it said it would.) The solution to this is almost always to implement the missing methods of the interface. A misspelled method name or incorrect parameter list is the usual cause, not that it should have been abstract!

A very common use of interfaces is for listeners. A listener is an object from a class that implements the required methods for that interface. You can create anonymous inner listeners, or implement the required interface in any class.

Interfaces are also used extensively in the data structures (Java Collections) package.

1.2. Classes versus Interfaces

Classes are used to represent something that has **attributes** (variables, fields) **and capabilities/ responsibilities** (methods, functions).
Interfaces are only about capabilities. For example, you are a human because you have the attributes of a human (class). You are a plumber because you have the ability of a plumber (interface). You can also be an electrician (interface). You can implement many interfaces, but be only one class.



This analogy fails in one way however. Capabilities (methods) of a class are unchanging (if a class implements an interface, it is implemented for all instances), whereas the human skills we're talking about are dynamic and can be learned or forgotten. The analogy is flawed, as all analogies are, but it gives some idea of a distinction between classes and interfaces.

Interfaces replace multiple inheritance

A C++ class can have more than one parent class. This is called multiple inheritance. Managing instance variable

definitions in multiple inheritance can be really messy, and leads to more problems – e.g, the "Deadly Diamond of Death" – than solutions. In the nearby figure an example of this problem is shown. The figure shows four classes arranged in the diamond structure that creates the need for virtual inheritance. Both of the classes B and C inherit from class A. D multiply inherits from both B and C. Two problems arise from this. First, which implementation of the 'f' method
does D inherit? Should it inherit f() from B or f() from C? In C++ the answer turns out to be neither. f() in D must be declared and implemented. This eliminates the ambiguity, and certainly this simple rule could have been adopted in Java.
The second problem, however, is quite a bit more complicated. The class A has a member variable named i. Both classes B and C inherit this member variable. Since D inherits from both B and C we face an ambiguity. On the one hand, we might want i of B and i of C to be separate variables in D; thus creating two copies of A in D. On the other hand we might want a single copy of A in D so that only i of A exists in D.
For this reason Java designers chose to allow only one parent class, but allow multiple interfaces. This provides most of the useful functionality of multiple inheritance, but without the difficulties.

1.3. Implementing an Interface

You may implement as many interfaces in a class as you wish; just separate them with commas. For example,

```
// Note:
//   ActionListener requires defining actionPerformed(...)
//   MouseMotionListener requires defining mouseMoved(...) and mouseDragged(...).

public class MyPanel extends JPanel implements ActionListener, MouseMotionListener {
    public void actionPerformed(ActionEvent e) {
        /* Method body */
    }
    public void mouseDragged(MouseEvent me) {
        /* Method body */
    }
    public void mouseMoved(MouseEvent me) {
        /* Method body */
    }
    // Everything else in this class.
}
```

It is common for a panel that does graphics and responds to the mouse to implement its own mouse listeners (but not action listeners) as above.

1.4. Declaring an interface

For simple programs you are more likely to use an interface than define it. Here is what the **java.awt.event.ActionListener** interface definition looks something like the following.

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

1.5. Initializing fields in interfaces

Fields defined in interfaces are automatically static and final. These cannot be "blank finals," but they can be initialized with nonconstant expressions. For example:

```
// Initializing interface fields with non-constant initializers.
import java.util.*;

public interface RandVals
{
  Random rand = new Random();
  int randomInt = rand.nextInt(10);
  long randomLong = rand.nextLong() * 10;
  float randomFloat = rand.nextLong() * 10;
  double randomDouble = rand.nextDouble() * 10;
}
```

Since the fields are static, they are initialized when the class is first loaded, which happens when any of the fields are accessed for the first time. Here's a simple test:

```
public class TestRandVals
{
  public static void main(String[] args) {
    System.out.println(RandVals.randomInt);
    System.out.println(RandVals.randomLong);
    System.out.println(RandVals.randomFloat);
    System.out.println(RandVals.randomDouble);
  }
}
```

The fields, of course, are not part of the interface but instead are stored in the static storage area for that interface.

1.6. Nesting interfaces

Interfaces may be nested within classes and within other interfaces. This reveals a number of very interesting features:

```
// NestingInterfaces.java

class A
{
  interface B
  {
    void f();
  }
  public class BImp implements B
  {
    public void f() {}
  }
  private class BImp2 implements B
  {
    public void f() {}
  }
  public interface C
  {
    void f();
  }
  class CImp implements C
  {
    public void f() {}
  }
  private class CImp2 implements C
```

```
  {
    public void f() {}
  }
  private interface D
  {
    void f();
  }
  private class DImp implements D
  {
    public void f() {}
  }
  public class DImp2 implements D
  {
    public void f() {}
  }
  public D getD() { return new DImp2(); }
  private D dRef;
  public void receiveD(D d)
  {
    dRef = d;
    dRef.f();
  }
}

interface E
{
  interface G
  {
    void f();
  }
  // Redundant "public":
  public interface H
  {
    void f();
  }
  void g();
  // Cannot be private within an interface:
  //! private interface I {}
}

public class NestingInterfaces
{
  public class BImp implements A.B
  {
    public void f() {}
  }
  class CImp implements A.C
  {
    public void f() {}
  }
  // Cannot implement a private interface except
  // within that interface's defining class:
  //! class DImp implements A.D {
  //!  public void f() {}
  //! }
  class EImp implements E
  {
    public void g() {}
  }
  class EGImp implements E.G
  {
```

```
    public void f() {}
  }
  class EImp2 implements E
  {
    public void g() {}
    class EG implements E.G
    {
      public void f() {}
    }
  }
  public static void main(String[] args)
  {
    A a = new A();
    // Can't access A.D:
    //! A.D ad = a.getD();
    // Doesn't return anything but A.D:
    //! A.DImp2 di2 = a.getD();
    // Cannot access a member of the interface:
    //! a.getD().f();
    // Only another A can do anything with getD():
    A a2 = new A();
    a2.receiveD(a.getD());
  }
}
```

The syntax for nesting an interface within a class is reasonably obvious, and just like non-nested interfaces, these can have public or package-access visibility. You can also see that both public and package-access nested interfaces can be implemented as public, package-access, and private nested classes.
As a new twist, interfaces can also be private, as seen in A.D (the same qualification syntax is used for nested interfaces as for nested classes). What good is a private nested interface? You might guess that it can only be implemented as a private inner class as in DImp, but A.DImp2 shows that it can also be implemented as a public class. However, A.DImp2 can only be used as itself. You are not allowed to mention the fact that it implements the private interface, so **implementing a private interface is a way to force the definition of the methods in that interface without adding any type information** (that is, without allowing any upcasting).
The method getD( ) produces a further quandary concerning the private interface: It's a public method that returns a reference to a private interface. What can you do with the return value of this method? In main( ), you can see several attempts to use the return value, all of which fail. The only thing that works is if the return value is handed to an object that has permission to use it—in this case, another A, via the receiveD( ) method.
Interface E shows that interfaces can be nested within each other. However, the rules about interfaces — in particular, that **all interface elements must be public** — are strictly enforced here, so an interface nested within another interface is automatically public and cannot be made private.
NestingInterfaces shows the various ways that nested interfaces can be implemented. In particular, notice that when you implement an interface, you are not required to implement any interfaces nested within. Also, private interfaces cannot be implemented outside of their defining classes.

1.7. Some Java predefined interfaces

1.6.1. The Clonable interface

The **Cloneable** interface is an example of a Java predefined interface:

- It does not contain method headings or defined constants
- It is used to indicate how the method **clone** (inherited from the **Object** class) should be used and redefined

- The method **Object.clone()** does a bit-by-bit copy of the object's data
- If the data is all primitive type data or data of immutable class types (such as **String**), then this is adequate. This is the simple case
- If the data in the object to be cloned includes instance variables whose type is a mutable class, then the simple implementation of **clone** would cause a *privacy leak*
- When implementing the **Cloneable** interface for a class like this:
  - First invoke the **clone** method of the base class **Object** (or whatever the base class is)
  - Then reset the values of any new instance variables whose types are mutable class types. This is done by making copies of the instance variables by invoking *their* clone methods
  - Note that this will work properly only if the **Cloneable** interface is implemented properly for the classes to which the instance variables belong, and for the classes to which any of the instance variables of the above classes belong, and so on and so forth

1.6.2. The comparable interface

- The **Comparable** interface is in the **java.lang** package, and so is automatically available to any program. It has only the following method heading that must be implemented:
  public int compareTo(Object other);
- It is the programmer's responsibility to follow the semantics of the **Comparable** interface when implementing it. The method **compareTo** must return
  - A negative number if the calling object "comes before" the parameter other
  - A zero if the calling object "equals" the parameter other
  - A positive number if the calling object "comes after" the parameter other
- If the parameter **other** is not of the same type as the class being defined, then a **ClassCastException** should be thrown
- Almost any reasonable notion of "comes before" is acceptable
  - In particular, all of the standard less-than relations on numbers and lexicographic ordering on strings are suitable
- The relationship "comes after" is just the reverse of "comes before"
- Other orderings may be considered, as long as they are a *total ordering*
- Such an ordering must satisfy the following rules:
  - (*Irreflexivity*) For no object **o** does **o** come before **o**
  - (*Trichotomy-antisymmetry*) For any two object **o1** and **o2**, one and only one of the following holds true: **o1** comes before **o2**, **o1** comes after **o2**, or **o1** equals **o2**
  - (*Transitivity*) If **o1** comes before **o2** and **o2** comes before **o3**, then **o1** comes before **o3**
- The "equals" of the **compareTo** method semantics should coincide with the **equals** method if possible, but this is not absolutely required
- Note: Both the **Double** and **String** classes implement the **Comparable** interface
  - Interfaces apply to classes only
  - A primitive type (e.g., **double**) cannot implement an interface

1.6.3. The Enumeration interface

- An object that implements the Enumeration interface generates a series of elements, one at a time.
  - an *iterator*
  - successive calls to the **nextElement()** method return successive elements of the series
  - **hasMoreElements()** tests if this enumeration contains more elements.
- Example, to print all elements of a vector *v*:

```
for (Enumeration e=v.elements(); e.hasMoreElements() ;)
{
    System.out.println(e.nextElement());
}
```

1.6.4. An example for Coneable and Comparable

Refer to the Employee example presented at the lectures.

```
abstract class Employee implements Comparable
 {
    // instance variables
   private String name;
   public Employee(String name)
   {
      this.name = name;
   }
   public String getName()
   {
      return name;
   }
   public int compareTo(Object other)
   {
      Employee e = (Employee) other;
     return name.compareTo(e.name);
   }

   public abstract double calculatePay();
}
```

```
abstract class Employee implements Cloneable, Comparable
 {
    // instance variables
    private String name;
    public Employee(String name)
   {
      this.name = name;
   }
   public String getName()
   {
      return name;
   }
   public int compareTo(Object other)
   {
      Employee e = (Employee) other;
      return name.compareTo(e.name);
   }
   public Object clone()
   {
       Cloneable theClone = new Employee(this.name);
       // Initialize theClone. No need here as String is the only variable and is immutable
       return theClone;
    }
   public abstract double calculatePay();
}
```

2. Lab Tasks

2.1. Study and understand the provided text and code.

2.2. Create an interface containing three methods, in its own package. Implement the interface in a different package.

2.3. For the following code, Sandwich.java, create an interface called FastFood (with appropriate methods) and change Sandwich so that it also implements FastFood.

```
// Sandwich.java
class Meal
{
  Meal() { System.out.println("Meal()"); }
}
class Bread
{
  Bread() { System.out.println("Bread()"); }
}
class Cheese
{
  Cheese() { System.out.println("Cheese()"); }
}
class Lettuce
{
  Lettuce() { System.out.println("Lettuce()"); }
}
class Lunch extends Meal
{
  Lunch() { System.out.println("Lunch()"); }
}
class PortableLunch extends Lunch
{
  PortableLunch() { System.out.println("PortableLunch()");}
}
public class Sandwich extends PortableLunch
{
  private Bread b = new Bread();
  private Cheese c = new Cheese();
  private Lettuce l = new Lettuce();
  public Sandwich()
  {
    System.out.println("Sandwich()");
  }
  public static void main(String[] args)
  {
    new Sandwich();
  }
}
```

2.4. Create three interfaces, each with two methods. Inherit a new interface from the three, adding a new method. Create a class by implementing the new interface and also inheriting from a concrete class. Now write four methods, each of which takes one of the four interfaces as an argument. In **main( ),** create an object of your class and pass it to each of the methods.