OOPS THROUGH JAVA

Interfaces

Interfaces

- In order to work with a class, you need to understand the public methods
  - methods, return types,…
  - after you instantiate, what can you do with it?
- The implementation details are irrelevant to using the class
  - you don’t have to know how things are done inside the class

Interfaces

- A Java *interface* is a collection of abstract methods and constants
- An *abstract method* is a method header without a method body
- An abstract method can be declared using the modifier `abstract`, but because all methods in an interface are abstract, usually it is left off
- An interface is used to establish a set of methods that a class will implement

Interfaces

interface is a reserved word

None of the methods in an interface are given a definition (body)

```
public interface Doable
{
   public void doThis();
   public int doThat();
   public void doThis2 (float value, char ch);
   public boolean doTheOther (int num);
}
```

A semicolon immediately follows each method header

Interfaces

- An interface cannot be instantiated
- Methods in an interface have public visibility by default
- A class formally implements an interface by:
  - stating so in the class header
  - providing implementations for each abstract method in the interface
- If a class asserts that it implements an interface, it must define all methods in the interface

Interfaces

```
public class CanDo implements Doable
{
    public void doThis ()
    {
        // whatever
    }

    public void doThat ()
    {
        // whatever
    }

    // etc.
}
```

implements is a reserved word

Each method listed in Doable is given a definition

Interfaces

- A class that implements an interface can implement other methods as well
- In addition to (or instead of) abstract methods, an interface can contain constants
- When a class implements an interface, it gains access to all its constants

Comparison with Inheritance

- Interfaces don't define any method actions… you need to fill in all the details
- It essentially just gives a basic collection of method names
- Not a strict hierarchy
- Important: can implement several Interfaces
- Can use this to "fake" multiple inheritance

Interfaces

- A class can implement multiple interfaces
- The class must implement all methods in all interfaces listed in the header

```
class ManyThings implements
  interface1, interface2
{
    // all methods of both interfaces
}
```

Example:The Comparable Interface

- Any class can implement `Comparable` to provide a mechanism for comparing objects of that type
- Specifically, implementing `Comparable` means that you need a method `CompareTo`

```
if (obj1.compareTo(obj2) < 0)
   System.out.println ("obj1 is less
  than obj2");
```

The Comparable Interface

- It's up to the programmer to determine what makes one object less than another
- For example, you may define the `compareTo` method of an `Employee` class to order employees by name (alphabetically) or by employee number
- The implementation of the method can be as straightforward or as complex as needed for the situation

Requiring Interfaces

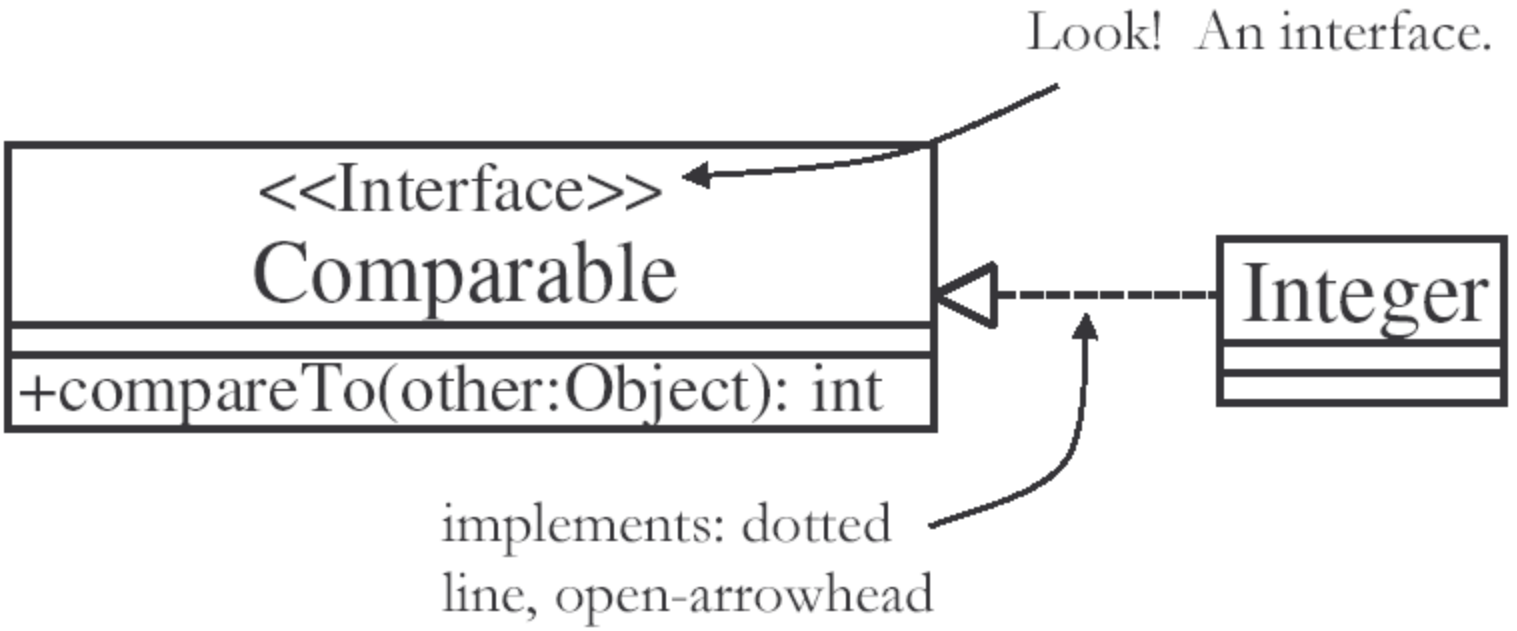
- Interface names can be used like class names in the parameters passed to a method

```
public boolean isLess(Comparable a,
                      Comparable b)
{
    return a.compareTo(b) < 0;
}
```

- Any class that "implements `Comparable`" can be used for the arguments to this method

Interfaces in UML

- Interfaces are easy to spot in class diagrams



Interfaces

- You could write a class that implements certain methods (such as `compareTo`) without formally implementing the interface (`Comparable`)
- However, formally establishing the relationship between a class and an interface allows Java to deal with an object in certain ways
- Interfaces are a key aspect of object-oriented design in Java

Built-in Interfaces

- The Java standard library includes lots more built-in interfaces
  - they are listed in the API with the classes
- Examples:
  - `Clonable` – implements a `clone()` method
  - `Formattable` – can be formatted with `printf`

The Iterator Interface

- As we discussed in Chapter 5, an iterator is an object that provides a means of processing a collection of objects one at a time
- An iterator is created formally by implementing the `Iterator` interface, which contains three methods
- The `hasNext` method returns a boolean result – true if there are items left to process
- The `next` method returns the next object in the iteration
- The `remove` method removes the object most recently returned by the `next` method

The Iterator Interface

- By implementing the `Iterator` interface, a class formally establishes that objects of that type are iterators
- The programmer must decide how best to implement the iterator functions
- Once established, the for-each version of the `for` loop can be used to process the items in the iterator

Collections

- `Collection` is a general interface for any type that can store multiple values
- Any object `c` that implements `Collections` has these methods
  - `c.add(e)`
  - `c.remove(e)`
  - `c.size()`

Collection Sub-Interfaces

- Interfaces that are derived from `Collection`
- `Set`: unordered, can't add the same object twice
- `List`: ordered, adds new methods
  - `get(i)`: get the i^{th} element
  - `set(i,e)`: set the i^{th} element to `e`

Collection Implementations

- Also in the standard library: many good implementations of these interfaces
- List: `ArrayList, Stack, LinkedList`
- Sets: `HashSet, TreeSet`
- Each implementation has some differences… suitable for particular problems
  - e.g. additional methods, different type restrictions, etc.

Example: Pairs

- A class to represent a pair (x,y) of values
- Both values represented with `Double`

```
class Pair
{
    Double x, y;

    public Pair(double x, double y)
    {
        this.x = new Double(x);
        this.y = new Double(y);
    }
}
```

Example: Pairs

- Want to be able to compare..

```
class Pair implements Comparable<Pair>
{
….
      public int compareTo(Pair other)
      {
            if(this.x.equals(other.x))
                  return this.y.compareTo(other.y);
            else
                  return this.x.compareTo(other.x);
      }
}
```

Implementing versus Inheriting

- Implementing an Interface is very similar to inheriting a class

```
class MyClass implements MyInterface {…}
```

  - Takes everything from MyInterface and puts it in MyClass
  - Except **all** the methods must be implemented here
  - No previous implementations to fall back on

Interfaces vs. Abstract Classes

- Similarities
  - neither can be instantiated
  - both can be used as the starting point for a class
- Differences
  - A class can contain implementations of methods
  - A class can implement many interfaces, but only one class

Comparison

- In order of “abstractness”:
  - Interface
    - no method implementations
    - can’t be instantiated
  - Abstract class
    - some method implementations
    - can’t be instantiated
  - Non-abstract class
    - all methods implemented
    - can be instantiated