Interfaces

Working with a Class

- In order to use a class, you need to understand the public methods.
  - names, parameters, return types, and what they do.
  - i.e. once you instantiate, what can you do with it?
- The implementation details are irrelevant to using the class.
  - you don't have to (and shouldn't) care about what's going on inside, just what it does.
  - Critical for separate debugging.

Common Methods

- Classes that represent similar items often share some common methods.
  - e.g. anything that can be sorted must implement the `compareTo()` method.
  - e.g. all of the "collections" implemention `add(x)` [insert new item] and `contains(x)` [in collection?]
  - collections include `ArrayList`, `Vector`, `Set`, …
- Often, we need "anything with `compareTo()`".

Interfaces

- An "interface" is a description of public methods
  - … their names, arguments and return types.
  - A class can "implement" several different interfaces.
- The `Comparable` interface describes the `compareTo()` method.
  - Sorting requires a class that implements the `Comparable` interface.

Abstract Methods

- The methods in an interface are "abstract".
  - They contain no implementation (or body: `{…}`), just argument types and return type.
  - They must be implemented in any class that implements the interface.
- So, you can't instantiate an interface.
  - Usage of methods is described, but there is no definition of their behaviour.

Implementing Interfaces

- The class definition only needs to indicate the interfaces it implements:

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

- … and then give definitions for the relevant methods.

Requiring Interfaces

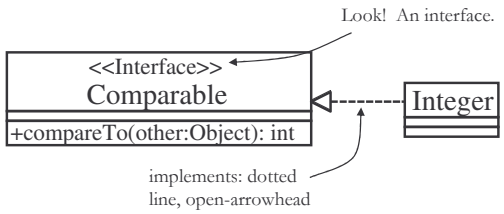
- To specify "any type that implements an interface" as an argument, use it as a type:

```
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 UML:



Built-In Interfaces

- The Java standard library contains many interfaces that can be used.
  - In the reference, they are listed along with the classes in each package.
- Examples:
  - `Clonable`: implements a `.clone()`
  - `Formattable`: can be formatted with `printf`

Collections

- The standard library has several interfaces and classes for "collections".
- `Collection` is a general interface for any type that can store multiple values.
  - Any object `c` that implements `Collection` has:

```
c.add(e)
c.remove(e)
c.size()
```

Collection Subinterfaces

- Interfaces that are derived from `Collection`:
- `Set`: unordered, can't contain duplicate elements
  - `c.add(e)` does nothing if `e` already in `c`.
- `List`: ordered, duplicates allowed. Adds methods relevant to ordered collections:
  - `c.get(i)`: get element at position `i`.
  - `c.set(i,e)`: set element at position `i` to `e`.

Collection Implementations

- Also in the standard library: many good implementations of these interfaces.
- Lists: `ArrayList`, `Stack`, `LinkedList`
- Sets: `HashSet`, `TreeSet`
- Each implementation has some differences.
  - different type restrictions, extra methods, running time for various operations, etc.

Using Interfaces

- The built-in interfaces cover a lot of common tasks.
- It's often useful to formally `implement` the corresponding interfaces.
  - This allows you to substitute your type anywhere the interface is required.
- Examples: list stored on-disk instead of in memory; set from database keys; …

Example

Pairs

- A class to represent a pair of values: (*x*, *y*)
  - 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);
    }
    …
}
```

Comparable 1

- The old way (before Java 5.0): all of the interfaces specify any object:

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

Comparable to any object

Must cast to a `Pair` so we can treat it like one. Bad if it wasn't a `Pair` & can't be checked until runtime.

Comparable 2

- The new way (in Java 5.0+): give a type parameter.

```
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);
        }
    }
}
```

Comparable to a `Pair`

Argument must be a Pair, but still satisfies interface.