CMSC 131
Object-Oriented Programming I

Interfaces

Dept of Computer Science
University of Maryland College Park

This material is based on material provided by Ben Bederson, Bonnie Dorr, Fawzi Emad, David Mount, Jan Plane
Overview

- Interfaces
- Polymorphism
Java Interfaces

- A **Java Interface** is a formal way for a class to **promise** to implement certain methods. We say that a class **implements** an interface if it provides these methods.
- The term **interface** should not be confused with the term interface used in API (Application Programmer Interface) and in GUI (Graphical User Interface).
- **Interface:**
  - Is defined by the keyword **interface** (rather than **class**).
  - It defines **methods** (as many as you like), but does **not** give **method bodies** (the executable statements that make up the method).
Defining a Java Interface:

- A Java interface is a collection of method declarations.
- These declarations are abstract, which means we do not supply the body of the method.

```java
public interface Y {
    public void someMethod(int z);
    public int anotherMethod();
}
```

- These methods are usually public, since they are expected to be part of an object’s public interface.
- Notice that an interface is not a class. For example, you cannot create an instance using “new Y”.
- Notice we cannot define instance variables (although we can define constants).

How to create them in Eclipse?

Example: animalExample package
Implementing an Interface:

- A class is said to “implement” an interface if it provides definitions for these methods.
- To inform Java that a class implements a particular interface $Y$, we add “implements $Y$” after the class name:

  ```java
  public class X implements Y {
    // ...(instance data and other methods)...
    public void someMethod(int z) { /* give implementation here */ }
    public int anotherMethod() { /* give implementation here */ }
  }
  ```

- Now, we may use an $X$ any place that an object of type $Y$ is expected.

Notice a class implementing an interface can implement additional methods

Notice that a class can implement several interfaces

**Example:** animalExample package
Motivation for Interfaces

- **Two Opposing Goals**, which Java programmers must deal with:
  - *Strong typing* and *General-Purpose Functions*

- **Strong Typing**: In strongly typed languages, like Java, the type of every variable must be specified. This makes debugging much easier.

- **General-Purpose Functions**: We would like to write methods that can be applied to many different types. For example, methods for *sorting*, computing *maximum* and *minimum*, etc. that can work with ints, doubles, Strings, etc. Advantages:
  - Less Coding
  - Less likely to have typos
  - Easier maintenance of code

- **The Problem**: Strong typing implies that, for example, to write a sorting function, we need to specify the types of the parameters (int, double, String, etc.). This makes it *impossible to write a generic sorting function*. It would seem that we need to implement many sorting functions (*sortInts()*), *sortDoubles()*, *sortStrings()*, *sortDates()*, *sortRationals()*, ...)

- **The Solution**: How can we solve the problem? By using Interfaces!
Java Interfaces

- **How it works:** Suppose you want to write a sorting method for objects of some class X. Sorting requires that you be able to compare the relative values of objects (<, >, <=, >=, ==).
  - You implement a **general-purpose sorting method**, using a comparison method (e.g., `compareTo()`).
  - The user of your sorting function **defines this comparison method** (`compareTo()`) for objects of class X.
  - Now it is possible to **invoke** your general sorting method on objects of class X.

- **To make this work:** Java needs to provide some mechanism for general-purpose functions (like sort) to specify **what behavior they require** from specific classes (like X). This is the purpose of a Java interface.
Comparable Interfaces

- The **Comparable** interface specifies a method called `compareTo` that takes an object as a parameter and returns a negative integer, zero, or a positive integer as the current object is less than, equal to, or greater than the specified object.
- Have we seen classes implementing this interface? Yes!
  - **String**
  - **Integer**
  - **Double**
  - All primitive wrapper classes implement **Comparable**
- By using interfaces a function like `Collections.sort()` can sort an `ArrayList` of objects that implement the `Comparable` interface. For example, an `ArrayList` of `Integers`, of `Strings`, etc.
- Can `Collections.sort()` sort an `ArrayList` of your own objects (e.g., `ArrayList` of `Cars`?) Yes! Just make the `Car` class implement the **Comparable** interface
- Through the **Comparable** interface we can have a general sorting function
  
  [http://download.oracle.com/javase/6/docs/api/java/lang/Comparable.html](http://download.oracle.com/javase/6/docs/api/java/lang/Comparable.html)

- **Example**: `Sorting.java`
- **Example**: `SortingCars.java`
Using an **interface** we can create one variable that can reference objects different types (i.e. Comparable variable referencing Integers, Strings or Cats; UMStudent variable referencing CSMajor, CEMajor or PsychMajor)

This form of “generalization” is called **polymorphism**
- Hallmark of OO languages
- Allows application of same code to objects of different types
- Polymorphism: “A variable that takes on many shapes.”

**Interfaces**: one mechanism Java provides for polymorphism

Interfaces allow us to define an IS-A relationship
- Dog is an Animal
- Not every Animal is a Dog