Interfaces

Defining an Interface
- Defining a Java Interface
  - A Java interface is a collection of method declarations
  - These declarations are abstract, which means that we do not supply the body of the method.
  - 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
- 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:
    - 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!

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., Array List of Cars?) Yes! Just make the Car class implement the Comparable interface.
- Through the Comparable interface we can have a general sorting function [http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html](http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html)
- **Example**: Sorting.java
- **Example**: SortingCars.java
- **NOTE**: You may not use Collections.sort() for your Poker project

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 (\(<\), \(\leq\), \(>\), \(\geq\), \(==\))
  - 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.

Polymorphism

- Using an interface we can create one variable that can reference objects of 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

http://docs.oracle.com/javase/8/docs/api/java/lang/Comparable.html
**Multiple Inheritance**

- **Motivation**: There are many situations where a simple class hierarchy is not adequate to describe a class’ structure.
- **Example**: Suppose we have our class hierarchy of university people and we also develop a class hierarchy of athletic people.

  ![Class Hierarchy Diagram]

- **StudentAthlete**: Suppose we want to create an object that inherits all the elements of a Student (admission year, GPA) as well as all the elements of an Athlete (sport, amateur-status).

- Can we define a StudentAthlete by inheriting all the elements from both Student and Athlete?

  ```java
  public class StudentAthlete extends Student, extends Athlete { ...
  ```

  - **Alas, no. At least not in Java**

  - **Multiple Inheritance**:
    - Building a class by extending multiple base classes is called multiple inheritance.
    - It is a very powerful programming construct, but it has many subtleties and pitfalls. (E.g., if Athlete and Student both have a name instance variable and a toString() method, which one do we inherit?)
    - Java does not support multiple inheritance. (Although C++ does.)
    - In Java a class can be extended from only one base class.
    - However, a class can implement any number of interfaces.

**“Faking” Multiple Inheritance with Interfaces**

- Java lacks multiple inheritance, but there is an alternative.
- What **public methods** do we require of an Athlete object?
  - String getSport(): Return the athlete’s sport.
  - boolean isAmateur(): Does this athlete have amateur status?

- We can define an interface Athlete that contains these methods:

  ```java
  public interface Athlete {
    public String getSport();
    public boolean isAmateur();
  }
  ```

- Now, we can define a StudentAthlete that **extends** Student and **implements** Athlete.

- **StudentAthlete** can be used:
  - Anywhere that a **Student object is expected** (because it is **derived** from Student).
  - Anywhere that an **Athlete object is expected** (because it implements the public interface of Athlete).

- So, we have effectively achieved some of the goals of multiple inheritance, by using Java’s single inheritance mechanism.
Common Uses of Interfaces

- Interfaces are flexible things and can be used for many purposes in Java:
  - A work-around for Java’s lack of multiple inheritance. (We have just seen this.)
  - Specifying minimal functional requirements for classes (This is its principal purpose.)
  - For defining groups of related symbolic constants. (This is a somewhat unexpected use, but is not uncommon.)

Using Interfaces for Symbolic Constants

- In addition to containing method declarations, interfaces can contain constants, that is, variables that are public final static. Sometimes interfaces are used just for this purpose.

Interface Hierarchies

- Inheritance applies to interfaces, just as it does to classes. When an interface is extended, it inherits all the previous methods.