Interfaces

CS 3: Computer Programming in Java
Objectives

- Introduce interfaces
- Get abstract with abstract methods
- See how we implement and extend interfaces
- Look at how we can perform multiple inheritance using interfaces
- Finish with a polymorphic sorting method
Interfaces in Java

- The fundamental unit of programming in Java is a class, but in object-oriented programming, the fundamental design unit is a reference type.
- Interfaces are a way to define reference types without defining a class.
  - Adds to Java’s power of object-oriented programming.
Interfaces and Classes in Java

- Interfaces define reference types in abstract form as a collection of public method headers and static constants
  - Contains no implementation of methods
  - Really is an expression of pure design

- Classes are a mixture of design and implementation
  - Can implement interfaces by using the keyword ‘implements’
  - EXAMPLE: If I design an interface called ElCaminoCollegeInterface, then get a class Student to implement it, class Student would be written as follows...
    ```java
    public class Student implements ElCaminoCollegeInterface {
        /* code */
    }
    ```
Interfaces and Classes in Java (2)

- Classes that implement an interface are agreeing to a design contract enforced by the interface method and static constants.
- Method names inside the interface are “holy” (i.e. they cannot be changed by the implementing subclasses).
  - The expectation is if “you send me an object name X, I will execute the method name Y”, with the implementation of the method being done in the subclasses.
More on Interfaces

- Interfaces can contain zero or more abstract methods and/or public static final fields
  - Interfaces with no methods in them can be considered marker interfaces
- Interfaces are also stored in .java files, just the way classes are
- Rules on interfaces
  ```java
  public interface SomeInterface {
    // Can contain zero or more public abstract methods
    // Can contain zero or more public static final fields
    // Cannot contain private or protected fields or methods
    // Cannot contain instance or class variables
  }
  ```
Abstract Method

- **ABSTRACT METHOD**: provides the prototype of a method, which is a header without a body
- **Syntax**
  
  ```
  [Access Specifier] [Other Specifier] abstract returnType methodName ([Argument List]);
  ```
- **Only required fields**
  - Return type
  - Method name
  - Argument list
Abstract Method (2)

- Can be specified in classes, as well as interfaces
  - Just need the keyword ‘abstract’ attached in front
- Most important thing about abstract methods is that they have no method body
Implementing and Extending Interfaces

- Interfaces can be implemented by or extended by another interface
- A class can implement more than one interface
- Class implementing an interface can implement the inherited abstract methods in any manner in which the designer of the class chooses to do so

**Syntax**

```java
public class SomeClass implements SomeInterface {
    // Inherits all the methods and static constants from SomeInterface
    // Must provide implementation (code) for all the inherited methods or else be declared
    // as an abstract class explicitly using the keyword ‘abstract’
}
```
public interface MyInterface
{
    void printHello();
}

class MyClass implements MyInterface
{
    public void printHello()
    {
        System.out.println("Hello from implemented method of interface MyInterface");
    }
}
public class DriverMyClass
{
    public static void main(String[] args)
    {
        MyClass MC1 = null;
        MyClass MC2 = new MyClass();
        MC2.printHello();
    }
}
About the Interface Example

- Cannot comment out printHello in class MyClass or else a compile error occurs
  - Due to the fact that the class MyClass adheres to a contract in implementing MyInterface that it would provide a body for the method printHello, whose header is designed in MyInterface

- Now, if we tried to change MyClass to be an abstract class using the syntax...
  ```java
  public abstract MyClass implements MyInterface {}
  ```
  - MyClass MC1 = null; will compile, but MC2 will not because we cannot create an instance of an abstract class
Interfaces Extending Other Interfaces

- Interfaces can have inheritance relationships between themselves as well
- An interface can be “extended” from one or more interfaces
- Syntax
  ```java
  public interface SomeInterface extends Interface1
  {
  }
  public interface SomeInterface extends Interface1, Interface2
  {
  }
  ```
public interface Door
{
    public void open();
    public void close();
}

public class CarDoor implements Door
{
    public void open()
    {
        System.out.println("Watch traffic. Enter the car.");
    }
    public void close()
    {
        System.out.println("Look out. Closing the door.");
    }
}
public class TestDoor
{
    public static void main(String[] args)
    {
        Door Instance1 = new CarDoor();
        Instance1.open();
        CarDoor Instance2 = new CarDoor();
        Instance2.close();
    }
}
The Door class has two main processes: open the door and close the door.

Therefore, two abstract methods created, open() and close(), which are both void methods and take no arguments.

Now, lots of objects in the world have doors (e.g. homes, cars, trucks).

To a degree, the behaviors of all doors is the same (they all open or close).

In this sense, the interfaces design behavior of objects.

Once a class implements an interface, it also develops an inheritance relationship with the implemented interface (e.g. Car Door is-a Door, Home Door is-a Door).
Inheritance Relationship Between Door and CarDoor
About the Inheritance Relationship Between Door and CarDoor

- CarDoor is a subclass or derived class from interface Door
  - A reference of type Door can hold the address of object of type CarDoor
  - With this, the line `Door Instance1 = new CarDoor();` compiles fine
  - Instance1 can be used to call any of the methods inherited from interface Door by the class CarDoor
    - Here, `Instance1.open();` would execute the code inside the `open()` method in CarDoor class
  - Implementing an interface also provides an additional reference type for a class
Multiple Inheritance Using Java Interfaces

- Java interfaces can provide a limited form of multiple inheritance in the sense that more than one reference type can represent any class that implements more than one interface.
C++ Inheritance

- C++ allows classes to be derived as a result of multiple inheritances.
- The class StudentVoter may inherit the fields and virtual functions from all of its super classes.
  - This, at times, may cause ambiguities and special care is needed to remove them.
Java, however, would not allow the inheritance shown in the previous slide if the entities shown are coded as classes.

One allowable scenario in Java is that entities Person and Voter are declared as interfaces and then Student and StudentVoter can be declared as classes.
Advantages/Limitations of Java Inheritance vs. C++ Inheritance

- The advantage of such multiple inheritance is that an object of StudentVoter Class may be represented by a reference of following types: Object, Person, Student, Voter, and StudentVoter
  - This is one of the key advantages of multiple inheritance
- The limitation however is that, unlike C++, in Java, Person and Voter being interfaces, they cannot contain any protected and instance fields
  - Java, in this sense, only allows “behavioral” multiple inheritance
Class/Interface Diagrams for Person, Student, Voter, and StudentVoter
public interface Person
{
    String getName();
    int getAge();
}

public interface Voter extends Person
{
    String PARTY1 = "Republican";
    String PARTY2 = "Democrat";
    String PARTY3 = "Liberatarian";
    String PARTY4 = "Independent";
    String PARTY5 = "None";

    String getParty();
}
public class Student implements Person {
    protected String Name;
    protected int age;
    protected double gpa;

    // Chained constructors
    public Student() {
        this("");
    }

    public Student(String Init_Name) {
        this(Init_Name, 0);
    }

    public Student(String Init_Name, int init_age) {
        this(Init_Name, init_age, 0.0);
    }
}
public Student(String Init_Name, int init_age, double init_gpa) {
    this.Name = Init_Name;
    this.age = init_age;
    this.gpa = init_gpa;
}

// Implementation of inherited abstract methods
public String getName() {
    return this.Name;
}

public int getAge() {
    return this.age;
}
public double getGpa()
{
  return this.gpa;
}

public String toString()
{
  String Str = "The name is = " + this.Name + "\n";
  Str += "The age = " + this.age + "\n";
  Str += "The GPA = " + this.gpa + "\n";
  return Str;
}
public class StudentVoter extends Student implements Voter
{
    private int lastvote; //records the year of last vote 0 for new voter
    private String Party;

    //Constructors
    public StudentVoter()
    {
        this("");
    }
    public StudentVoter(String Init_Name)
    {
        this(Init_Name,0);
    }
}
public StudentVoter(String Init_Name, int init_age)
{
    this(Init_Name, init_age, 0.0);
}

public StudentVoter(String Init_Name, int init_age, double init_gpa)
{
    this(Init_Name, init_age, init_gpa, 0);
}

public StudentVoter(String Init_Name, int init_age, double init_gpa, int init_lastvote)
{
    this(Init_Name, init_age, init_gpa, init_lastvote, "");
}
public StudentVoter(String Init_Name, int init_age, double init_gpa, int init_lastvote, String Init_Party) {
    super(Init_Name, init_age, init_gpa);
    this.lastvote = init_lastvote;
    if(Init_Party.equals(StudentVoter.PARTY1))
        this.Party = PARTY1;
    else if(Init_Party.equals(StudentVoter.PARTY2))
        this.Party = PARTY2;
    else if(Init_Party.equals(StudentVoter.PARTY3))
        this.Party = PARTY3;
    else if(Init_Party.equals(StudentVoter.PARTY4))
        this.Party = PARTY4;
    else if(Init_Party.equals(StudentVoter.PARTY5))
        this.Party = PARTY5;
    else {
        System.out.println("Bad party name. Setting party to none.");
        this.Party = PARTY5;
    }
}
// Implementation of inherited abstract methods; only Voter interface methods need be implemented
// since the Person interface methods are already implemented by the super class Student
public String getParty()
{
    return this.Party;
}

// Helper methods
public String toString()
{
    String Str = super.toString();
    Str += "The year of last vote = " + this.lastvote + \n;  
    Str += "The party affiliation is = " + this.Party + "\n";
    return Str;
}

public int getLastVote()
{
    return this.lastvote;
}
Tests the diamond inheritance formed by the interfaces Person and Voter and classes Student and StudentVoter

```java
public class TestDiamond {
    public static void main(String[] args) {
        StudentVoter STVoter1 = new StudentVoter("Elliot John", 23, 3.92, 1998, "Republican");

        System.out.println(STVoter1);
        System.out.println("The party of " + STVoter1.getName() + " is " + STVoter1.getParty());
        System.out.println("The age of " + STVoter1.getName() + " is " + STVoter1.getAge());
        System.out.println("The year of last vote by " + STVoter1.getName() + " is " + STVoter1.getLastVote());
        System.out.println("The gpa of " + STVoter1.getName() + " is " + STVoter1.getGpa());
    }
}
```
// Showing Multiple inheritance behavior
// A reference of type Person can point to both Student and Student Voter
Person Person_Gen = new Student("Mary Smith", 29, 3.0);
Person Stu1 = Person_Gen;
System.out.println("n" + Person_Gen);
System.out.println("The age of " + Person_Gen.getName() + " is " + Person_Gen.getAge());
    /*uncommneting the code below will cause compile error
     Because getGpa cannot be called with Person Type reference
     Person_Gen must be cast into a Student type reference.*/
    /*System.out.println("The gpa of " + Person_Gen.getName() + " is " +
     Person_Gen.getGpa());*/
    System.out.println("The gpa of " + Person_Gen.getName() + " is " + ((Student)(Person_Gen)).getGpa());
System.out.println("\n" + Person_Gen);

Person [ ] Array = {STVoter1,Stu1,Person_Gen};
//Calling the polymorphic sort method
selectionSort(Array);
//Print the sorted array
System.out.println("Printing the sorted array - sorted based on name.");
for (int ind=0; ind<Array.length; ind++)
   System.out.println (Array [ind]);
/** The method selection sort accepts an array of type Person and sorts the array alphabetically based on name. Reference to sorted array can be used to print the array in the caller method */

public static void selectionSort(Person[] list)
{
    Person largest = null;
    int length = list.length;
    int passCount;
    int L_Index;
    int S_Index;
}
for ( passCount = 0 ; passCount < length ; passCount++)
{
    largest = list[0];
    L_Index = 0; //Find the index of largest
    for ( S_Index=0; S_Index < (length - passCount) ; S_Index++)
    {
        if(((largest.getName()).compareTo(list[S_Index].getName()))<0)
        {
            L_Index = S_Index;
            largest = list [S_Index];
        }
    }
    list [ L_Index ] = list [ length - (passCount+1) ]; //swap now
    list [ length - (passCount+1) ] = largest ;
}
The power of polymorphism and multiple inheritance is shown in the form of polymorphic methods `selectionSort`, which takes an array of `Person` type as an argument and sorts the array alphabetically based on first name.

Notice that the beauty of this method is that `Person` array passed to it may have all elements as:
- `Student` type
- `StudentVoter` type
- Mixture of `Student` and `StudentVoter` both

Still, the method will sort the array correctly based on the name in alphabetical order.

We see that all class objects represented by their respective objects behave as expected.
Next, we create an array of type Person, and put three objects in it, which are respectively, of type StudentVoter, Student, and StudentVoter.

Their references are mixed type.

The first one has a reference of type StudentVoter (STVoter1).

The second one of type Student (Stu1).

The third of type Person (Person_Gen).

Person array can hold any of the reference types shown.
The method selectionSort is a standard sorting method, which works based on a well-known comparison based sort principle of array sorting technology:

- The method looks for the object, which has the largest member (in this case the name) and bubbles it to the end of the array.
- The maximum number of passes required to sort the array are one less than the array size.

We can see very clearly from the results that the polymorphic method selectionSort works nicely as it takes an array of mixed Student and StudentVoter type objects and sorts them in the alphabetical order by the name.

Becoming able to write polymorphic methods is another main advantage of inheritance in Java.
Summary

- Interfaces
  - Define reference types without defining a class
  - Method names inside the interface are “holy”
    ```java
    public interface SomeInterface
    {
        // Can contain zero or more public abstract methods
        // Can contain zero or more public static final fields
        // Cannot contain private or protected fields or methods
        // Cannot contain instance or class variables
    }
    ```

- Abstract methods
  - Provides the prototype of a method
Implement and extend interfaces
- Interfaces can be implemented by or extended by another interface
- A class can implement more than one interface
- Class implementing an interface can implement the inherited abstract methods in any manner in which the designer of the class chooses to do so

Multiple inheritance using interfaces
- Java requires a combination of interfaces and classes

Polymorphic sorting method
- Being able to pass in many types of data into the same method