This section of lecture notes is devoted to the discussion of Interfaces in Java:

Thanks to Prof. Satish Singhal for allowing me to include these excerpts from his notes on Java Programming:

**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. This adds to Java’s power of object-oriented programming. Interfaces define reference types in abstract form as a collection of public method headers and static constants. **Interface contains no implementation of methods; therefore, we can say that an interface is an expression of pure design.** On the other hand, the classes are a mixture of design and implementation. Java classes can implement interfaces by using the keyword implements. For example if I design an interface called ElCaminoCollegeInterface and then get a class Student to implement it, then class Student would be written as follows:

```java
public class Student implements ElCaminoCollegeInterface { /*code*/ }
```

The classes that implement interface are agreeing to a design contract enforced by the interface method headers and static constants. The method names inside the interface are “holy”. They cannot be changed by the implementing subclasses. This is crucial in Graphical Interface Programming in Windows, where when the user clicks on an icon, or presses a key on the keyboard, the Operating System fires an event object to the program and then in order to respond to the user action, the program must execute a certain method. There is a
contract with the operating system that if “you send me an object name X, I will execute the method name Y”. This contract cannot be violated. Implementation of methods named inside the interfaces enforces this contract. Interface can contain zero or more abstract methods and/or public static final fields (see sidebar below). The interfaces with no methods in them can be considered marker interfaces. Interfaces are also stored in .java files, just the way classes are.

```java
public interface SomeInterface
{
    /*Can contain zero or more public abstract methods
    and zero or more public static final fields.
    Cannot contain private or protected fields or methods.
    Cannot contain instance or class variables.*/
}
```

What is an abstract method? An abstract method simply provides the proto-type of a method, which is a header without a body. The syntax of writing an abstract method is as follows:

```
[Access Specifier] [Other Specifier] abstract returnType methodName
([Argument List]);
```

In above list the entities inside square brackets are optional. Inside an interface only return type, method name and argument list are required. Abstract methods can be specified with in classes also, in which case they would need the keyword abstract attached in front. Most important thing is that abstract methods have no method body.

Interface can be implemented by a class or be extended by another interface. A class can implement more than one interface. The class implementing an interface can implement the inherited abstract methods in any manner in which the designer of the class chooses to do so. The syntax of implementing and writing an interface is as follows:

The class implementing a java interface uses keyword implements to implement it. (see below).
A class implementing an interface must either provide implementation of all of its methods or be declared as an abstract class\(^1\). For example let us take a system of an interface MyInterface, a class implementing it MyClass, and a driver class DriverMyClass to test MyClass (Listing 10.9)

```java
public interface MyInterface
{
    void printHello();
}

public class MyClass implements MyInterface
{
    public void printHello()
    {
        System.out.println("Hello from implemented method of "+" interface MyInterface");
    }
}
```

```java
public class DriverMyClass
{
    public static void main(String[] args)
    {
        MyClass MC1 = null;
        MyClass MC2 = new MyClass();
        MC2.printHello();
    }
}
```

A class implementing an interface must either provide implementation of all of its methods or be declared as an abstract class\(^1\). For example let us take a system of an interface MyInterface, a class implementing it MyClass, and a driver class DriverMyClass to test MyClass (Listing 10.9)

```java
public interface MyInterface
{
    void printHello();
}

public class MyClass implements MyInterface
{
    public void printHello()
    {
        System.out.println("Hello from implemented method of "+" interface MyInterface");
    }
}
```

```java
public class DriverMyClass
{
    public static void main(String[] args)
    {
        MyClass MC1 = null;
        MyClass MC2 = new MyClass();
        MC2.printHello();
    }
}
```

\(^1\) Abstract class cannot be instantiated, though it can be used as a reference type. We discuss abstract classes shortly.
In implementing interface MyInterface the class MyClass adheres to a contract that it would provide a body for the method printHello whose header is designed in MyInterface. Therefore inside the main method of driver class DriverMyClass, or elsewhere the MyClass will behave like a normal class in the sense that it can be instantiated and its public methods can be called. The execution of main method in Listing 10.9 will print to console a string “Hello from implemented method of interface MyInterface”.

However, as soon as the method printHello in class MyClass is commented out, all statements inside the main will give a compile error shown in the Figure 10.17.

The main thrust of compile errors is error #2, where compiler indicates that class MyClass is breaching the contract with interface MyInterface by not overriding or implementing the method printHello. If one declares the class MyClass abstract using the syntax:

```java
public abstract MyClass implements MyInterface
{
}
```

then the statement

```
MyClass MC1 = null;
```

will compile, however one still cannot instantiate the abstract class MyClass to create instance MC2. Therefore classes not overriding or implementing the methods from their implemented interface must be declared abstract explicitly by using keyword abstract.

Interfaces can have inheritance relationships between themselves as well. An interface can be “extended” from one or more interfaces. The syntax is as follows:
Note that keyword “extends” is used when an interface extends another interface.

We show an example of writing a java interface and its implementation in Listing 10.10 (A-C) below.

```java
public interface Door
{
    public void open( );
    public void close( );
}
//Listing 10.10A

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");
    }
}
//Listing 10.10B

public class TestDoor
{
    public static void main( String[ ] args )
    {
        Door Instance1 = new CarDoor( );
    }
    //Listing 10.10C
}
```

Only abstract methods!!! Use of access specifier public is optional and can be left out.

Provides implementation for all the methods inherited from interface Door!

A reference of type Door can point to an object of type CarDoor because CarDoor is-a Door.
As one knows that a door has two main processes – open the door or close the door. Therefore the interface Door is given two abstract methods open and close, and in this case, both are void method and take no arguments. Now, lots of objects in world have doors. Homes have doors. Cars have doors. Trucks have doors. To the degree that behaviors of all doors (in cars, homes, trucks) is same (they all open or close), the classes for these objects can all implement interface Door. **In this sense the interfaces design behavior of objects!** Of course all classes implementing Door interface can implement their interface methods open and close in their own way. For example the message printed by the close method of class HomeDoor may be different from the one for class CarDoor as per its needs.

Once a class implements an interface it also develops an inheritance relationship with the implemented interface. For example we can say that a Car Door is-a Door. Or a Home Door is-a Door. And as discussed earlier under polymorphism, the inherited class instance can be pointed to by a reference of type of its super class. Therefore the reference type Door can hold the address of a CarDoor type object created on heap (first statement in the main method in Listing 10.10C). Figure 10.18 shows the inheritance relationship between interface Door and class CarDoor.
FIG. 10.18
The diagram simply means that CarDoor is a subclass or derived class from interface Door. If contract to implement methods open and close is kept then a reference of type Door can hold the address of object of type CarDoor. Therefore the first statement in the main method of class Test Door given below compiles fine.

```java
Door Instance1 = new CarDoor();
```

Now the Instance1 (which is a Door type reference) can be used to call any of the methods inherited from interface Door by the class CarDoor. Thus Instance1.open would execute the code inside the open method in CarDoor class. Implementing an interface, apart from forming a contractual relationship of implementing interface methods, also provides an additional reference type for a class. The output of Listing 10.10C is shown in Figure 10.19.
Multiple Inheritance Using Java Interfaces
Java interfaces can provide a limited form of multiple inheritance, in the sense that more than one reference types can represent any class that implements more than one interface. Figure 10.20 below shows the most difficult case of multiple inheritance, which is also called the “diamond” multiple inheritance.

In C++ Person, Student, Voter, and StudentVoter can all be classes!

C++ allows classes to be derived as a result of multiple inheritances. Therefore in C++, Person, Student, Voter and StudentVoter can all be classes. 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 Figure 10.20 if all 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 (Figure 10.21).
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. 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.

Codes for classes and interfaces in Figure 10.21 are shown in Listing 10.11. The UML diagram for Figure 10.21 is shown in the Figure 10.22.
As shown in the UML diagram, we put two methods in the interface `Person`: `getName()` and `getAge()`. These methods relate to the fact that each person will have an age or name. More methods may be added in the `Person` interface, but for the illustration purpose two methods include would suffice. The coded person interface is shown below in the Listing 10.11 Part A.

```java
public interface Person {
    String getName();
    int getAge();
}
```

Interface `Voter` extends interface `Person`. That means that the methods in the `Person` interface are also automatically inherited by the interface `Person`. We however include some static constant strings in the `Voter` interface to hard code the possible party a student voter may be member of. We also include a method `getParty()` in the `Voter` interface, which returns the party of the student voter as a string. The coded `Voter` interface is shown in the Listing 10.11 Part B.

```java
public interface Voter extends Person {
    String PARTY_1 = "Democratic";
    String PARTY_2 = "Republican"

    String getParty();
    String toString();
    int getLastVote();
    int main(String[] args);  
}
```
After coding the two interfaces, now we can code the necessary classes. First, we code the class Student, which derives from interface Person. Class Student has name, age and gpa as fields, and we have shown all the necessary chained constructors (Listing 10.11 Part C). Since Person interface is implemented by the class Student, we provide the implementation of the methods getAge( ) and getName( ). We also code some helper methods such as getGpa( ), and toString( ). The method getGpa( ) is unique to class Student, as only a person, who is a student can have gpa. The toString( ) method overrides the corresponding method in the Object class. The code for the class Student is shown in the Listing 10.11 Part C.
this.gpa = init_gpa;

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

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

//Other helper methods
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;
}

Listing 10.11 Part C
The class StudentVoter extends Student and implements interface Voter. Note that since class Student already implemented the abstract methods from Person class, the class StudentVoter need not do that, unless it wishes to provide a specialized version of the interface methods by overriding them. The general rule that applies here is that if any of the super-classes implement the methods of an interface, they are already available to all sub-classes. However, the StudentVoter must implement the abstract method getParty( ) from the Voter interface. The StudentVoter class introduces two new fields, the date of last vote (lastvote) and Party of the StudentVoter (Party). This is because the StudentVoter inherits all the fields from the Student class, it in effect, has five fields and needs six chained constructors. It has helper methods such as getLastVote and over-rides the toString method from object class2. Listing 10.11 Part D shows the code for the class StudentVoter.

2 We discuss overriding the Object class methods in more detail later.
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;
        }
    }
}
Listing 10.11 Part D

The class TestDiamond is written to test the diamond inheritance formed by the interfaces and classes and is shown in Listing 10.11E. We create three objects, two of type StudentVoter and one of type Student. The references used to point to them are StudentVoter, Student and Person. In creation of all objects we call the constructor with largest number of arguments for that class. The object STVoter1 is of type StudentVoter. It can be printed directly by the System.out.println because the method toString() of StudentVoter class is overrides the object class toString method to correctly print the StudentVoter objects. The methods getName, getAge, getGpa, getParty are easily called using STVoter1 because STVoter1 is a StudentVoter Type reference pointing to a StudentVoter type object.

Next we create a Person type reference Person_Gen which points to a Student object. This is now polymorphism in play, as one can use a Person type reference to point to Student or StudentVoter object. The Person_Gen reference being a Person type can only be used to call the methods of Person interface on the object of type Student. Therefore one can only call methods getName and getAge. Attempt to call method getGpa will cause compile error, but if Person_Gen is cast into a reference of type Student, then a call to getGpa method can be made easily. We assign an additional reference to object being pointed to by Person_Gen. This additional reference is Student type called Stu1. Then we go ahead and use Person_Gen reference to point to a new object of StudentVoter type (name field for this object being “Barney Bush”). This is possible because a StudentVoter is also a Person type.
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("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("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 ;
    }
}

Listing 10.11 Part E

Polymorphic Sorting Method

The power of polymorphism and multiple inheritance is shown in Listing 10.11-E in
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. The results of test run based on Listing 10.11 Part E (TestDiamond class) are
shown in Figure 10.23. 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) and third of type
Person (Person_Gen). Person array can hold any of the reference types shown in
Figure 10.23.
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.