CMSC 202

Interfaces

Classes and Methods

- When a class defines its methods as public, it describes how the class user interacts with the method.
- These public methods form the class' **`interface`** .
- An abstract class contains one or more methods with only an interface – no method body is provided.
- Java allows us to take this concept one step further.

Interfaces

- An interface is something like an extreme abstract class.
- All of the methods in an interface are abstract – they have no implementations.
- An interface
  - has no instance variables.
  - Only defines methods.
  - is NOT a class.
  - is a type that can be satisfied by any class that implements the interface

Interfaces

- The syntax for defining an interface is similar to that of defining a class
  - Except the word `interface` is used in place of `class`

- An interface specifies a set of methods that any class that implements the interface must have
  - It contains method headings (and optionally static final constant definitions) only
  - It contains no instance variables nor any complete method definitions

Interfaces

- An interface and all of its method headings should be declared public
  - They cannot be given private, protected, or package access
- When a class implements an interface, it must make all the methods in the interface public.
- Because an interface is a type, a method may be written with a parameter of an interface type
  - That parameter will accept as an argument any class that implements the interface

Implementing an Interface

- To create a class that implements all the methods defined in an interface, use the keyword **implements**.
- Whereas **interface** defines the headings for methods that must be defined, a class that **implements** the interface defines how the methods work.

The Animal Interface

```
public interface Animal
{
  public void eat( );
}
```

Yes, animals do more than eat, but we’re trying to make this a simple example.

Interfaces

- To *implement an interface*, a concrete class must do two things:

1. It must include the phrase

```
implements Interface_Name
```

at the start of the class definition

   – If more than one interface is implemented, each is listed, separated by commas

2. The class must implement *all* the method headings listed in the definition(s) of the interface(s)

10/11

Implementing Animal

```
// Lion and Snake implement the required eat( ) method
public class Lion implements Animal
{
   public void eat()
        { System.out.println("Lions Devour"); }
}
public class Snake implements Animal
{
   public void eat()
        { System.out.println( "Snakes swallow whole"); }
}
```

Implementing Animal

```
// Dog implements the required eat( ) method and has
// some of its own methods and instance variables
public class Dog implements Animal {
   private String name;
   Dog(String newName)
        {name = newName;}
   public void eat()
        {System.out.println("Dog chews a bone");}
}

// Poodle is derived from Dog, so it inherits eat( )
// Adds a method of its own
public class Poodle extends Dog
{
   Poodle( String name )
        { super(name); }  // call Dog constructor

   public String toString( )
        { return "Poodle"; }
}
```

Implementing Animal

```
// Using classes that implement Animal
public class Jungle
{
   public static void feed( Animal a )
        { a.eat(); }

   public static void main( String[] args ){
        Animal[ ] animals = {
                new Lion( ),
                new Poodle( "Fluffy" ),
                new Dog( "Max" ),
                new Snake( )
        };
        for (int i = 0; i < animals.length; i++)
                feed( animals[ i ] );
   }
}
```

```
// --- Output
```

Lions Devour
Dog chews a bone
Dog chews a bone
Snakes swallow whole

Extending an Interface

- An new interface can add method definitions to an existing interface by **extending** the old

```
interface TiredAnimal extends Animal
{
   public void sleep( );
}
```

The TiredAnimal interface includes both eat( ) and sleep( );

Interface Semantics Are Not Enforced

- When a class implements an interface, the compiler and run-time system check the syntax of the interface and its implementation
  - However, neither checks that the body of an interface is consistent with its intended meaning
- Required semantics for an interface are normally added to the documentation for an interface
  - It then becomes the responsibility of each programmer implementing the interface to follow the semantics
- If the method body does not satisfy the specified semantics, then software written for classes that implement the interface may not work correctly

The `Comparable` Interface

- The **`Comparable`** interface is in the **`java.lang`** package, and so is automatically available to any program
- It has only the following method heading that must be implemented (note the Object parameter)

```
public int compareTo(Object other);
```

- It is the programmer's responsibility to follow the semantics of the **`Comparable`** interface when implementing it
- When implementing **`compareTo`**, you would of course overload it by using an appropriate parameter type

The `Comparable` Interface Semantics

- The method **`compareTo`** must return
  - A negative number if the calling object "comes before" the parameter other
  - A zero if the calling object "equals" the parameter other
  - A positive number if the calling object "comes after" the parameter other

- If the parameter **`other`** is not of the same type as the class being defined, then a **`ClassCastException`** should be thrown

The `Comparable` Interface Semantics

- Almost any reasonable notion of "comes before" is acceptable
  - In particular, all of the standard less-than relations on numbers and lexicographic ordering on strings are suitable
- The relationship "comes after" is just the reverse of "comes before"

compareTo for Person

```
public class Person implements Comparable
{
   private String name;
   ...
   public int compareTo( Object obj )
   {
       Person p = (Person)obj;
       return name.compareTo(p.name);
   }
   ....
}
```

If **obj** is not a **Person** object a **ClassCastException** will be thrown

Comparing the names using String's **compareTo** method

Using Comparable

```
public class NumTests {
   // find the smallest Integer in an array
   // Integer (implements Comparable )
   public static void findSmallest(Integer[] values) {
        int index = 0;          // index of smallest value

        for (int i = 1; i < values.length; i++)
        {
               if ( values[i].compareTo( values[index] ) < 0 )
                      index = i;
        }
        System.out.println("Index of smallest value is " + index);
   }
```

Using Comparable

```
// prints the index of the smallest Integer in an array
// Note use of Integer, not int
public class Test{
   public static void main( String[ ] args)
   {
        Integer[ ] values = {
                new Integer(144), new Integer(200), new Integer(99),
                new Integer(42),  new Integer(132) };
        NumTests.findSmallest(values);
   }
}
```

- But what if we wanted to operate on Floats, or Strings, or…
- Power comes from the fact that interfaces are also types

Using Comparable

```
public class NumTests {
   // find the smallest Integer in an array
   // Integer (implements Comparable)
   public static void findSmallest(Integer[]    values) {
        int index = 0;        // index of smallest value

        for (int i = 1; i < values.length; i++)
        {
                if ( values[i].compareTo( values[index] ) < 0 )
                        index = i;
        }
        System.out.println("Index of smallest value is " + index);
   }
```

Using Comparable

```
public class NumTests {
   // find the smallest  thing  in an array
   // Comparable is a type!
   public static void findSmallest(Comparable[] values) {
        int index = 0;          // index of smallest value

        for (int i = 1; i < values.length; i++)
        {
                if ( values[i].compareTo( values[index] ) < 0 )
                        index = i;
        }
        System.out.println("Index of smallest value is “ + index);
   }
```

Implementing Multiple Interfaces

- Recall the Animal interface from earlier

```
public interface Animal
{
   public void eat( );
}
```

- Define the Cat interface

```
public interface Cat
{
   void purr( );      // public by default;
}
```

`// since a Lion is an Animal and a Cat, Lion may wish`
`// to implement both interfaces`

```
public class Lion implements Animal, Cat
{
   public void eat( ) {System.out.println("Big Gulps");}
   public void purr( ) {System.out.println("ROOOAAAR!");}
}
```

Just separate the Interface names with a comma

Inconsistent Interfaces

- In Java, a class can have only one base class
  - This prevents any inconsistencies arising from different definitions having the same method heading
- In addition, a class may implement any number of interfaces
  - Since interfaces do not have method bodies, the above problem cannot arise
  - However, there are other types of inconsistencies that can arise

Inconsistent Interfaces

- When a class implements two interfaces:
  - Inconsistency will occur if the interfaces contain methods with the same name but different return types
- If a class definition implements two inconsistent interfaces, then that is an error, and the class definition is illegal