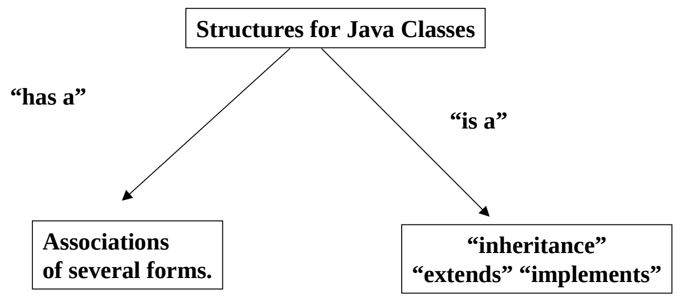
Interfaces and Abstract Classes



Inheritance among classes, use of the “extends” specification, was covered in CS303E. We will assume familiarity with use of the “extends” specification among classes and introduce interfaces and the “implements” specification.

One major motivation is “generic programming.” Generic programming is that method written for one object type can be applied for another object type.

Interfaces and Abstract Classes

Roadmap for Lecture

1. Definitions and Rules

interfaces, abstract classes, implements, etc.

2. Single Versus Multiple Inheritance

Student/Employee Example

3. Generic Programming with Abstract Classes

Sorting of arrays

4. Generic Programming with Interfaces.

Sorting of Tiles

Interfaces and Abstract Classes

Entities In Inheritance Relationships

Interface - A name and specification for a set of method signatures which is associated with that name.

[Class Modifiers] **interface** InterfaceName
[**extends** Interface1, Interface2 …]
{Interface Method Specifications}

```
public interface MyInterface
{ public void aMethod() }
public interface Comparable
{ public int compareTo(Object O) }
```

interfaces can not define instance variables but can define constants.
interfaces do not have constructors.
Instances of interfaces cannot be created.
Method signatures in interfaces cannot have the "static" attribute.

Interfaces and Abstract Classes

Class - An interface, a state and an implementation of the interface.
Abstract Class - An interface, a state and a partial implementation of the interface.

```
public abstract class MyClass
/* This class cannot be instantiated. */

public abstract class Message
{ private String sender;
 public message(String from)
        {sender = from;}
public abstract play();
public String getSender()
{return sender;}
}
```

Abstract classes cannot be instantiated.

Interfaces and Abstract Classes

Semantics of “extends” and “implements.”

***extends - classes*: The “extends” specification applied to class definitions states that the “extended” class “inherits” the instance variables and methods of the extending class.**
A class can “extend” or “inherit from” only one other class.

public class MyClass extends YourClass
/* The instance variables of YourClass are a part of the state of MyClass and the methods of YourClass can be invoked on instances of MyClass. */

***extends - interfaces* : The “extends” specification applied to interface definition states that the extended interface includes the method signatures defined in the extending interface.**
An interface can extend or “inherit from” any number of other interfaces.

public interface Comparable extends Clonable
/* Comparable includes the method signatures specified in Clonable. */

Interfaces and Abstract Classes

Implements - classes implement interfaces.

[ClassModifiers] **class** ClassName
[**extends** SuperClass]
[implements Interface1, Interface2, ….]
{Class MemberDeclarations, ClassMethodDeclarations}

```
interface student              interface employee
{float getGPA()                {float getSalary()
//other methods}               //other methods }

public class StudentEmployee implements student, employee
        {float getGPA(){ implementation of getGPA}
         float getSalary(){ implementation of getSalary}
        }
```

Note that a class can implement many interfaces.

Interfaces and Abstract Classes

Rules of Structuring

1. If class C_1 extends class C_2 then C_1 is a subtype of C_2.

2. If interface I_1 extends interface I_2 then I is a subtype of I_2.

3. If class C implements interface I then C is a subtype of I.

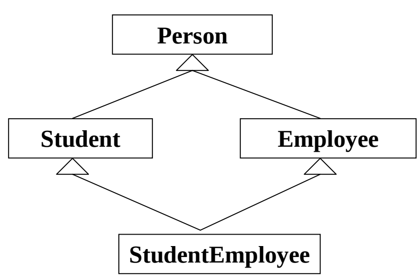
4. For every interface I, I is a subtype of Object.

5. For every type T, refernce or primitive type, T[] (array of type T) is a subtype of Object.

6. If type T_1 is a subtype of T_2 then T_1 is a subtype of T_2 [].

Interfaces and Abstract Classes

What is the role of Interfaces? Single versus multiple inheritance and Generic Programming



StudentEmployee inherits from both Student and Employee and both Student and Employee inherit from Person.

Interfaces and Abstract Classes

Public class Person
{ protected String name;
public String getName() {};
……….
}
public class Student extends Person
{protected float GPA;
public float getGPA(){}
…….}
public class Employee extends Person
{protected float salary;
public float getSalary();
……..
}
public class StudentEmployee extends Student, Employee
/* Illegal in Java - Why? */

Student inherits the name instance variable from Person,

Employee inherits the name instance variable from Person

StudentEmployee inherits the name instance variable from both Student and Employee.

Does StudentEmployee have two names?

Interfaces and Abstract Classes

The Interface Approach to Multiple Inheritance

interface Student

{public float getGPA()}

interface Employee

{public float getSalary()}

public class StudentImpl implements Student

{protected float GPA;

public float getGPA(){Implementation of getGPA.}

}

public class EmployeeImpl implements Employee

{protected float salary;

public float getSalary(){Implementation of getSalary}

}

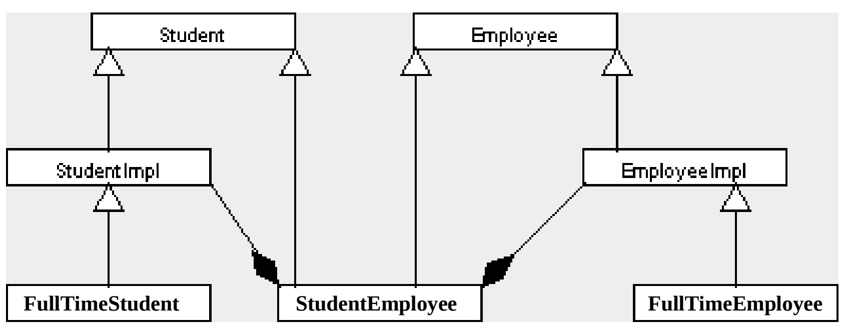
Interfaces and Abstract Classes

```
Public class StudentEmployee implements Student,
Employee
{ protected StudentImpl studentImpl;
  protected EmployeeImpl employeeImpl;
  public StudentEmployee()
        {studentImpl = new StudentImpl();
        employeeImpl = new EmployeeImpl();
        …..
        }
  public float getGPA()
        {return studentImpl.getGPA();}
/* StudentEmployee has delegated this method to be
implemented by class StudentImpl. */
  public float getSalary()
        {return employeeImpl.getSalary()}
/* StudentEmployee has delegated this method to be
implemented by class EmployeeImpl. */

………..
}
```


Interfaces and Abstract Classes

UML Diagram of Student- Employee Relationships



Interfaces and Abstract Classes

Generic Programming

1. A method written for one object type can be applied for another object type.
2. An Example: An array of objects of class type can be converted to an array of class Objects. For example, an Employee[] array can be passed to a method which expects an Object array.
3. An Example of “generic programming.

```
Static int find(Object[], a, Object key)
{   int i;
    for (i = 0; i < a.length, i++)
            if (a[i].equals(key)) return i;
            return  -1 //  Not found }

Employee[] staff = new Employee[10];
Employee harry = new Employee(…..);
staff[i] = harry;
int n = find(staff, harry);
```


Interfaces and Abstract Classes

Implementation of a Generic Sorting Algorithm using an abstract class.

```
abstract class Sortable
{  public abstract int compareTo(Sortable b);
/* Returns:  -1 if the invoking object instance is lower in the sorting order
                  than the argumentinstnace ,
                0 if the two instances are equal
               +1 if the invoking object instance is higher in the sorting order
                  than the argument instance.
}
```

Example - Sort an array of Employees

1. Derive Emplpyee from Sortable
2. Implement the CompareTo() method in Employee.
3. Call ArrayAlg.Shellsort() on an array of Employee object instances.

Interfaces and Abstract Classes

Demonstration of Generic Sort

```
import java.util.*;
import corejava.*;

public class EmployeeSortTest
{  public static void main(String[] args)
   {  Employee[] staff = new Employee[3];
      staff[0] = new Employee("Harry Hacker", 35000,
         new Day(1989,10,1));
      staff[1] = new Employee("Carl Cracker", 75000,
         new Day(1987,12,15));
      staff[2] = new Employee("Tony Tester", 38000,
         new Day(1990,3,15));
      ArrayAlg.shellSort(staff);
      int i;
      for (i = 0; i < staff.length; i++)
         System.out.println(staff[i]);
   }
}
```


Interfaces and Abstract Classes

```
public abstract class Sortable
{  public abstract int compareTo(Sortable b);
}
public class ArrayAlg
{  public static void shellSort(Sortable[] a)
   {  int n = a.length;
      int incr = n / 2;
      while (incr >= 1)
      {  for (int i = incr; i < n; i++)
         {  Sortable temp = a[i];
            int j = i;
            while (j >= incr
               && temp.compareTo(a[j - incr]) < 0)
            {  a[j] = a[j - incr];
               j -= incr;
            }
            a[j] = temp;
         }
         incr /= 2;
      }
   }
}
```

Interfaces and Abstract Classes

```
class Employee extends Sortable
{  public Employee(String n, double s, Day d)
   {  name = n;
      salary = s;
      hireDate = d;
   }

   public void raiseSalary(double byPercent)
   {  salary *= 1 + byPercent / 100;
   }

   public String getName() { return name; }

   public double getSalary() { return salary; }

   public String toString()
   {  return name + " " + salary + " " + hireYear();
   }
```


Interfaces and Abstract Classes

```
public int hireYear()
   {  return hireDate.getYear();
   }

   public int compareTo(Sortable b)
   {  Employee eb = (Employee)b;
      if (salary < eb.salary) return -1;
      if (salary > eb.salary) return 1;
      return 0;
   }

   private String name;
   private double salary;
   private Day hireDate;

}
```

Interfaces and Abstract Classes

Defining the abstract class and implementing the sorting algorithm was actually unnecessary.

The Java package java.util provides a class, Arrays, which defines and implements many useful operations on arrays including sorting and searching. Please look at the material on Arrays at

http://java.sun.com/products/jdk/1.2/docs/api/java/util/Arrays.html#sort(java.lang.Object[])

Take a look at the several methods defined in Arrays including the method public void sort(Object[] O) at

http://java.sun.com/products/jdk/1.2/docs/api/java/util/Arrays.html#sort(java.lang.Object[])

Then click on the link to the interface Comparable which we have already discussed once.

http://java.sun.com/products/jdk/1.2/docs/api/java/lang/Comparable.html

Interfaces and Abstract Classes

A sort() method defined in class Arrays in java.util.

public static void sort(Object[] a)

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements.

All elements in the array must implement the Comparable interface. Furthermore, all elements in the array must be mutually comparable (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array).

This sort is guaranteed to be stable: equal elements will not be reordered as a result of the sort. The sorting algorithm is a modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist). This algorithm offers guaranteed n*log(n) performance, and can approach linear performance on nearly sorted lists.

Parameters:

a - the array to be sorted.

Throws:

ClassCastException - if the array contains elements that are not mutually comparable (for example, strings and integers).

See Also:

Comparable

Interfaces and Abstract Classes

Implementation of a Generic Sorting Algorithm using interfaces.

Let us look at another example code in detail. This is an example where we cannot use an abstract class.

The class Tile models tiled windows on a screen desktop. Tiled windows are rectanges plus a z-order variable which governs the display order. Tiles with a larger z-order are displayed in front of those with a smaller z-order. We will need to keep the instances of Tile in sorted order by z-order.

The choices for keeping the instances of Tile in sorted order include:

1. Implementation of a sort algorithm in the Tile class

2. Have the Tile class inherit from one of the many Java classes which implement sort algorithms.

Number 2 sounds like much less work!

The Tile class inherits from the Rectangle class of the awt package. Therefore we cannot have it inherit from the abstract class Sortable. So Number 2 is not possible.

But interfaces solve this problem.

Interfaces and Abstract Classes

public interface Comparable

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

A class's natural ordering is said to be consistent with equals if and only if (e1.compareTo((Object)e2)==0) has the same boolean value as e1.equals((Object)e2) for every e1 and e2 of class C.

Interfaces and Abstract Classes

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals operation.

For example, if one adds two keys a and b such that (a.equals((Object)b) && a.compareTo((Object)b) != 0) to a sorted set that does not use an explicit comparator, the second add operation returns false (and the size of the sorted set does not increase) because a and b are equivalent from the sorted set's perspective.

Virtually all Java core classes that implement comparable have natural orderings that are consistent with equals. One exception is java.math.BigDecimal, whose natural ordering equates BigDecimals with equal values and different precisions (such as 4.0 and 4.00).

Interfaces and Abstract Classes

Method Summary
int compareTo(Object o)
Compares this object with the specified object for order.

Method Detail
public int compareTo(Object o)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure sgn(x.compareTo(y)) == -sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.)

The implementor must also ensure that the relation is transitive: (x.compareTo(y)>0 && y.compareTo(z)>0) implies x.compareTo(z)>0.

Finally, the implementer must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.

Interfaces and Abstract Classes

It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)).

Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

Parameters:

o - the Object to be compared.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws:

ClassCastException - if the specified object's type prevents it from being compared to this Object.

Interfaces and Abstract Classes

```
class Tile extends Rectangle implements Comparable
{  public Tile(int x, int y, int w, int h, int zz)
   {  super(x, y, w, h);
      z = zz;
   }

   public int compareTo(Object b)
   {  Tile tb = (Tile)b;
      return z - tb.z;
   }

   public String toString()
   {  return super.toString() + "[z=" + z + "]";
   }

   private int z;
}
```

Interfaces and Abstract Classes

```
import java.awt.*;
import java.util.*;
public class TileTest
{  public static void main(String[] args)
   {  Tile[] a = new Tile[20];
      int i;
      for (i = 0; i < a.length; i++)
         a[i] = new Tile(i, i, 10, 20,
            (int)(100 * Math.random()));
      Arrays.sort(a);
      for (i = 0; i < a.length; i++)
         System.out.println(a[i]);
   }
}
```


Interfaces and Abstract Classes

```
Tile[x=4,y=4,width=10,height=20][z=3]
Tile[x=2,y=2,width=10,height=20][z=4]
Tile[x=7,y=7,width=10,height=20][z=5]
Tile[x=0,y=0,width=10,height=20][z=7]
Tile[x=18,y=18,width=10,height=20][z=7]
Tile[x=10,y=10,width=10,height=20][z=11]
Tile[x=13,y=13,width=10,height=20][z=16]
Tile[x=11,y=11,width=10,height=20][z=24]
Tile[x=16,y=16,width=10,height=20][z=24]
Tile[x=9,y=9,width=10,height=20][z=25]
Tile[x=17,y=17,width=10,height=20][z=28]
Tile[x=14,y=14,width=10,height=20][z=33]
Tile[x=15,y=15,width=10,height=20][z=49]
Tile[x=6,y=6,width=10,height=20][z=52]
Tile[x=3,y=3,width=10,height=20][z=56]
Tile[x=5,y=5,width=10,height=20][z=56]
Tile[x=8,y=8,width=10,height=20][z=59]
Tile[x=19,y=19,width=10,height=20][z=66]
Tile[x=1,y=1,width=10,height=20][z=68]
Tile[x=12,y=12,width=10,height=20][z=69]
```