Interfaces and related issues

Reading for these lectures:
Weiss, Section 4.4 (The interface), p. 110.

no multiple inheritance with classes

```
public class Student {          public class Employee {
…                               …
}                               }

public class StudentEmployee
    extends Student, Employee { … }
```

not allowed.
doesn't work.

Such multiple inheritance has been tried in several languages, but it never works correctly. Trouble with ambiguity. E.g. what if method getName is defined in both superclasses but they yield different values? No good theory of such multiple inheritance has been developed yet, so Java does not allow it.

1

!!Interrupt!!!

Don't override variables. Don't do this:

```
public class C {
    public int x;
}

public class Sub extends C {
    public int x;
}
```

(Actually called **shadowing** the variable.)

Overriding methods is useful. We know of know compelling case where overriding a variable helps. Our rules for executing method calls do not take into account overriding variables. Forget completely about overriding variables, and never do it.

2

The interface

An **interface** contains the result types (or **void**) and signatures of some methods. The methods are called "abstract" because their bodies are not present; the bodies are replaced by semi-colons. The methods are automatically **public**.

```
public interface Comparable {
    // =  if this Object < ob  then a negative integer
    //    if this Object = ob, then 0
    //    if this Object > ob, then a positive integer
    int compareTo (Object ob);
}
```

no prefix (static, public, etc.)

abstract method: body replaced by ";"

The **signature** of a method consists of its name and its parameter declarations (enclosed in parentheses and separated by commas).

3

Implementing an interface

```
public interface Comparable {
    // =  if this Object < ob  then a negative integer
    //    if this Object = ob, then 0
    //    if this Object > ob, then a positive integer
    int compareTo (Object ob);
}
```

The "implements clause" in the class definition indicates that the class defines all the methods of the named interface. It's a syntactic error not to do so; the program won't compile.

```
public class Shape implements Comparable {
    …
    int compareTo(Ob ob) {
        body of method compareTo
    }
}
```

definition of compareTo

implements clause

4

Implementing method compareTo

In this case, the parameter of compareTo is expected to be an instance of Shape, because the method is defined in Shape

```
public class Shape implements Comparable {
    …
    // =  if this Object < ob  then a negative integer
    //    if this Object = ob, then 0
    //    if this Object > ob, then a positive integer
    //    (it's expected that ob is really a Shape)
    int compareTo(Object ob) {
        if (this.area() < ((Shape)ob).area())
            return -1;
        if (this.area() == ((Shape)ob).area())
            return 0;
        return 1;
    }
}
```

ob is cast to Shape

(in this case, Shapes are ordered by their area)

5

Why is the interface concept useful?

```
public class ArrayMethods {
    // = index of the max value in nonempty b
    public static int max(Comparable[] b) {
        int j= 0;
        // {invariant: b[j] is the max of b[0..i-1]}
        for (int i= 1; i != b.length; i= i+1) {
            if (b[i].compareTo(b[j]) > 0)
                j= i;
        }
        return j;
    }
}
```

max will find the max of any array b whose base class implements Comparable!

```
Shape[] s= new Shape[20];
Integer[] x= new Integer[100];
```

Fill s and x with values.

```
int maxs= ArrayMethods.max(s);
int maxx= ArrayMethods.max(x);
```

In Java 2, version 1.3.1, all primitive-type wrapper classes except Boolean implement Comparable.

6

An interface acts like a type, and casts to and from an interface are allowed!

```
public class ArrayMethods {
    // = index of the max value in nonempty b
    public static int max(Comparable[] b) {
        int j= 0;
        // {invariant: b[j] is the max of b[0..i-1]}
        for (int i= 1; i != b.length; i= i+1) {
            if (b[i].compareTo(b[j]) > 0)
                j= i;
        }
        return j;
    }
}
```

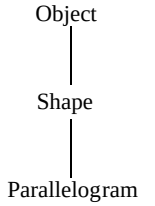
unnecessary cast because it widens

```
Shape[] s= new Shape[20];
Fill s with values.
int maxs= ArrayMethods.max((Comparable[]) s);
int maxs= ArrayMethods.max(s);
```

7

Class and interface hierarchies

```
public class Shape implements Comparable
    { … }
public class Parallelogram extends Shape
    implements Comparable, Comp2
    { … }
public interface Comparable { … }
public interface In1 { … }
public interface Comp2 extends In1 {… }
```

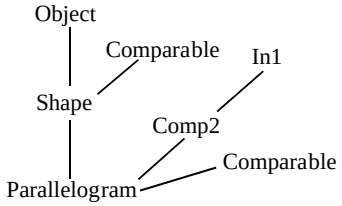


An interface, like a class, is handled like a type: can cast to and from an interface.

8

Class and interface hierarchies

```
public class Shape implements Comparable
    { … }
public class Parallelogram extends Shape
    implements Comparable, Comp2
    { … }
public interface Comparable { … }
public interface In1 { … }
public interface Comp2 extends In1 {… }
```

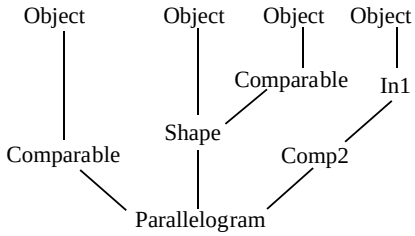


Upward cast: widening; done automatically when necessary

Downward cast: narrowing; not automatic

9

Class and interface hierarchies



Parallelogram p= **new** Parallelogram(); **legal**

Shape s= (Shape) p;	**legal**
(Comparable) p	**legal**
(Comparable) s	**legal**
(Comp2) p	**legal**
(Comp2) s	**illegal**
(In1) p	**legal**
(In1) s	**illegal**
((In1)p).equals(…)	**legal**

Note that Object acts like a super interface

In1 I= (In1) p; Using In1, can reference only names accessible in In1.

10

```
public class Arrays {
    public static int max(Comparable[] b) {
        int j= 0;
        // {invariant: b[j] is the max of b[0..i-1]}
        for (int i= 1; i != b.length; i= i+1) {
            if (b[i].compareTo(b[j]) **  >  0)
                j= i;
        }
        return j;
    }
}
public class Shape implements Comparable {
    int compareTo(Object ob) {
        Shape x= (Shape)ob;
        if (this.area() < x.area())   return -1;
        if (this.area() == x.area()) return 0;
        return 1;
    }
}
public class M {
    public static void main(…) {
        Shape[] s= new Shape[20];
        Fill s with values.
        int m= Arrays.max(s); L1:
    }
}
```

11

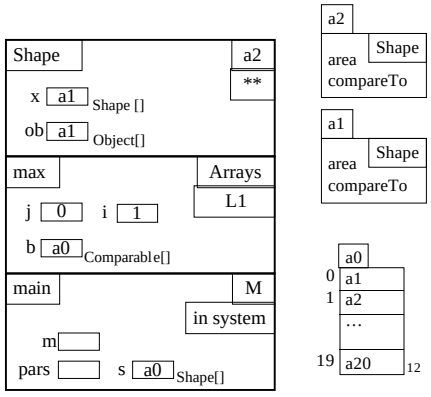
This slide shows the call stack for the call

Arrays.max(s);

on the previous slide, just after execution of

Shape x= (Shape)ob;

in method a2.compareTo.



12

Functors (function objects)

Interface Comparable doesn't fit all situations. For an array of integers, there are several ways to sort it -- ascending order, descending order, in order of distance from 0 (e.g. 0,1,-1, 2, -2, 3, -4, …), etc. We want to use the same sort method to sort the array in any order.

Solution: pass a comparison function to method sort:

```
// Sort array b using sort method f
public static void sort(int[] b, function f) {
    … if  f(b[i],b[j]) ...
}
```

illegal in Java!

```
public static void main(String[] pars) {
    int[] x= new int[50];    Fill array x with values;
    sort(x, greaterequal);
    sort(x, lessequal);
}

// = "x <= y"
public static boolean lessequal(int x, int y)
    {return x <= y;}

// = "x >= y"
public static boolean greaterequal(int x, int y)
    {return x >= y;}
```

13

Functors (function objects)

A function cannot be an argument, but an instance of a class that is guaranteed to contain a function can!

```
// A functor with boolean function compare(x,y)
public interface CompareInterface {
    // =   x <= y
    boolean compare(Object x, Object y);
}
```

An instance of interface is a functor: an instance with exactly one function defined it it.

```
// Sort array b using functor c
public static void sort(int[] b, CompareInterface c) {
    …
    if  c.compare(b[i],b[j]) ...
}
```

parameter c is guaranteed to contain function compare

14

Consequence of using a functor

One sort method can be used to sort an array of any base class; you provide the functor that does the comparing.

```
// A functor with boolean function compare(x,y)
public interface CompareInterface {
    // =  x  compared to  y
    boolean compare(Object x, Object y);
}

public class Less implements CompareInterface {
    // =  "x < y"
    public boolean compare(Object x, Object y)
       {return
         ((Integer )x).intValue() < ((Integer )y).intValue();}
}

public class Greater implements CompareInterface {
    // =  "x > y"
    public boolean compare(Object x, Object y)
       {return
         ((Integer )x).intValue() > ((Integer )y).intValue();}
}

// Sort array b using functor c
public static void sort(int[] b, CompareInterface c) {
    … if  c.compare(b[i],b[j]) ...
}
```

15

Consequence of using a functor

```
// A functor with boolean function compare(x,y)
public interface CompareInterface {
    // =  x  compared to  y
    boolean compare(Object x, Object y);
}

public class Less implements CompareInterface {
    // =  "x < y"
    public boolean compare(Object x, Object y)
       {return …; }
}

public class Greater implements CompareInterface {
    // =  "x > y"
    public boolean compare(Object x, Object y)
       {return …; }
}

// Sort array b using functor c
public static void sort(int[] b, CompareInterface c) {
    … if  c.compare(b[i],b[j]) ...
}

public static void main(String[] pars) {
    int[] x= new int[50];    Fill array x with values;
    sort(x, new Less());
    sort(x, new Greater());
}
```

16

Nested classes

Using functors as we have been doing tends to lead to a proliferation of classes (and files that contain them), like Less, and Greater, which may be used only once:

```
public static void main(String[] pars) {
    int[] x= new int[50];    Fill array x with values;
    sort(x, new Less());
}
```

may be only creation of instance of Less in the whole program!

```
public class Less implements CompareInterface {
    // = "x < y"
    public boolean compare(Object x, Object y)
       {return …; }
}
```

Nested classes, as illustrated on the next slide remove the need for so many files.

17

Nested classes

```
public class MainClass {

    public static void main(String[] pars) {
        int[] x= new int[50];    Fill array x with values;
        sort(x, new Less());
    }

    private static class Less
              implements CompareInterface {
        // = "x < y"
        public boolean compare(Object x, Object y)
        {return …; }
    }
}
```

Less, defined using **static**, is called a nested class. It is nested inside class MainClass. It could have any visibility modifier; it need not be **private**.

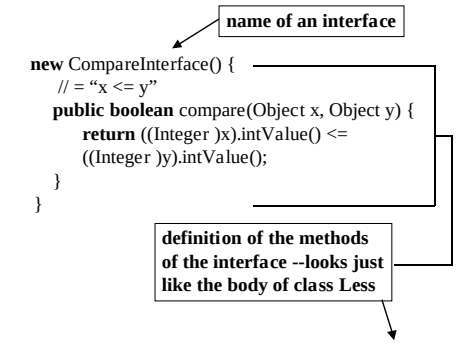
Because Less is declared within MainClass, it is a static component of MainClass, and it can reference any other static components.

A component like Less is called a nested class ONLY if it is **static**.

18

Anonymous classes

Even the use of a nested class forces us to name a class that may be used only once. There is a way to write a class that is going to be used only once, without giving it a name --it's an **anonymous** class. Here's an example:




```
public class Less implements CompareInterface {
    // = "x < y"
    public boolean compare(Object x, Object y)
      {return …; }
}
```

19

Use of an anonymous class

```
public static void main(String[] pars) {
    int[] x= new int[50];    Fill array x with values;
    sort(x, new CompareInterface() {
              // = "x <= y"
              public boolean compare(Object x, Object y) {
                   return ((Integer )x).intValue() <=
                             ((Integer )y).intValue();
              }
         }
        );
}
```

create instance of the anonymous class

Use interface name as the name of a class. Put in body the desired implementation of function compare.

no longer needed

```
public class Less implements CompareInterface {
    // = "x < y"
    public boolean compare(Object x, Object y)
      {return …; }
}
```

20

Iterators

Reading: Weiss, Sect 6.1,
Sect 6.2, the Iterator pattern,
Sect 6.3.2, Interface Iterator

We will see one way that Java features allow for the reuse of program components.

Definition: A data structure: representation of data together with operations that manipulate the date.

21

Iterators

Loop to process elements of an array:

```
int j= 0;
while (j != b.length) {
    Process b[j];
    j= j+1;
}
```

Loop to process prime numbers less than 100:

```
int j= 2;
while  (j < 100) {
    Process j;
     Set j to the next prime number
}
```

There is a pattern to process any sequence v. Write an interface for that pattern.

```
while (there exists another item to process) {
    Get the next item;
    Process the item
}
```

Write an interface that can be used to implement classes for iterating over any sequence.

22

Interface java.util.Iterator
(only in version 1.2 and 1.3)

```
// An Iterator allows processing of items of a sequence
public interface Iterator {
    // = "there is another item to process"
    public boolean hasNext();

    // = the next item to process. Can be called only once
    // per item. Throws an exception if no more items exist
    public Object next() throws NoSuchElementException;

    // Remove the last element returned by the Iterator.
    // Throw an UnsupportedOperationException if not
    // implemented.
    // Throw an IllegalStateException if next has not yet
    // been called or remove was already called for the item
    public void remove() throws
                UnsupportedOperationException,
                IllegalStateException;
}
```

We won't deal with method remove for the moment.

23

Interface java.util.Enumeration
(in all versions 1.2 and 1.3)

```
// An Enumerator allows processing of items of a sequence
public interface Iterator {

    // = "there is another item to process"
    public boolean hasMoreElements();

    // = the next item to process. Can be called only once
    // per item. Throws an exception if no more items exist
    public Object nextElements()
            throws NoSuchElementException;
}
```

Java would rather you use interface Iterator than Enumeration.

24

Example of use of interface Iteration

```
import java.util.*;
// An instance is an iterator over a prefix of the primes
public class PrimeIterator implements Iterator{
    int n;       // produce primes less than n
    int p= 2;  // the next prime to produce
    // Constructor: an instance for primes less than n
    public PrimeIterator(int n) { this.n= n; }

    // = there is another prime to process
    public boolean hasNext() { return p < n; }

     // = the next item to process. Can be called only once
    // per item. Throws an exception if no more items exist
    public Object next() throws NoSuchElementException {
      if ( p >= n)  throw new NoSuchElementException();
      int returnValue= p;
      p= p+1;
      while (!isPrime(p))   {  p= p+1;}
      return new Integer(returnValue);
    }

     // Remove last element returned by Iterator. Unsupported
    public void remove() {throw …; }

      // = "p is a prime". Precondition: p >= 2
      public boolean isPrime(int p) { … }
}
```

25

Example of use of class PrimeIterator

```
import java.util.*;

public class Example {
    public static void main(String[] pars) {
        PrimeIterator pi=  new PrimeIterator(50);
        while (pi.hasNext()) {
            Integer i=  (Integer)pi .Next();
            System.out.println(i);
        }
}
```

or

```
public class Example {
    public static void main(String[] pars) {
        PrimeIterator pi=  new PrimeIterator(50);
        while (pi.hasNext()) {
            System.out.println(pi);
        }
}
```

26

Example of use of class KeyboardIterator, which is given on next slides

```
import java.util.*;

public class Example {
    public static void main(String[] pars) {
        System.out.println("Type numbers into keyboard");
        KeyboardIterator ki= new KeyboardIterator();
        while (ki.hasNext()) {
            Integer i= (Integer)ki.next();
            System.out.println("integer is: " + i);
            System.out.flush();
        }
    }
}
```

27

Class KeyboardIterator

```
import java.io.*;  import java.util.*;
// Iterator for processing a seq of nonzero ints, one per line,
// from keyboard (terminated by 0). Assume that keyboard
// is not open at the time. The items are of class Integer.
// hasNext should be called first, and calls to hasNext
// and next should alternate
public class KeyboardIterator implements Iterator{
      BufferedReader br;   // Link to keyboard
      int p;               // The last integer read (1 if none)
      boolean toHasNext; // Next call should be to hasNext
                           // (and not to next)

      // Constructor: an instance that reads from the keyboard
      public KeyboardIterator() {
          InputStreamReader isr=
                new InputStreamReader(System.in);
          br= new BufferedReader(isr);
          p= 1;   toHasNext= true;
      }
```

28

Class KeyboardIterator (continued)

```
// = there is another input integer to process
public boolean hasNext() {
  if (!toHasNext)
      throw new RuntimeException("hasNext was called" +
                  " when next should have been called");
  toHasNext= false;
  p= readInt();
  if (p != 0) return true;
  try {
    br.close();
  }
  catch (IOException ex) {
    System.out.println("IO error reading from keyboard");
    System.exit(0);
  }
  return false;
}
```

29

Class KeyboardIterator (continued)

```
// = the next item to process, as an Integer. Call only
// once per item. Throw exception if no more items exist
public Object next() throws NoSuchElementException {
  if (toHasNext)
      throw new RuntimeException("hasNext called " +
                  "when next should have been called");
  if (p == 0)
      throw new RuntimeException("no more elements");
  toHasNext= true;
  return new Integer(p);
}

// Not supported. Does nothing.
public void remove() {  }

// Read and return the next integer from br  (the keyboard)
public int readInt()
  {  As in recitation handout  }
}
```

30

Some uses of interfaces

Functors. We are able to define a class that is guaranteed to define a certain function. Allows us, for example, to define a single sort procedure that will sort any array whose base class implements compareTo.

Iterators. We are able to define a pattern that can be used to define classes for iterating over (or enumerating) any finite sequence of values --a range of integers, a range of prime numbers, the values of an array, the lines of a file, the integers typed on the keyboard, and so on. Later, we see how to define iterators over (or enumerators of) various "data structures".

Define "abstract datatypes". A data type is a set of values together with a set of operations on them. Example: a stack, which is a list of values with three operations: push a value onto (the top of) the stack, pop the top element of the stack, and a test for emptiness. We can define such a datatype using an interator and then implement it in various ways. The assignment for this topic gives you practice in this.

31