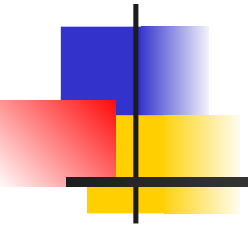


Lexical Ordering and Sorting

These slides refer to interfaces.

Lexical Ordering

Many kinds of objects can be ordered.

Numbers ("<" defines an ordering):

```
1
2
2.01
2,980,000
```

Strings (character collation defines an ordering):

```
Ark
act
car
cat
zebra
```

Ordering using compareTo()

- Java classes defines the lexical ordering of objects using a method named compareTo( ).
- Examples: String, Date, Double, all have compareTo

```
String s1 = "Cat";
String s2 = "Dog";
// which comes first in dictionary: Cat or Dog?
if ( s1.compareTo( s2 ) < 0 ) {
   // s1 comes before s2
} else if ( s1.compareTo( s2 ) > 0 ) {
   // s1 comes after s2
} else {
   // s1 & s2 have the same lexical order
}
```

Sort Data in an Array

`java.util.Arrays` has utility methods for arrays.

One method is: **`Arrays.sort( array[ ] )`**

```
// Sort an array of Strings
// String has a compareTo() that defines order
String[] words = {"dog","cat","ant","DOGS","BIRD"};
Arrays.sort( words );
```

`words` array

```
dog
cat
ant
DOGS
BIRD
```

Arrays.sort( )

Result:

```
words[0] = "BIRD"
words[1] = "DOGS"
words[2] = "ant"
words[3] = "cat"
words[4] = "dog"
```

Sort part of an Array

If the array is not full, you can sort just the part of the array containing values you want.

Use:

```
Arrays.sort(array[ ], start_index, end_index)
```

```
// sort elements 0 to count (exclusive)
int count = 5;  // we have 5 words to sort
Arrays.sort( words, 0, count );
```

This sorts only the elements

```
words[0] words[1] ... words[count-1]
```

Arrays.sort( ) can sort almost anything

Arrays.sort() can sort any many kinds of objects:

- array of Date
- array of String
- array of BigDecimal

- How does Arrays.sort know what lexical order to use?
- It calls the objects' own compareTo( ) method.
- This makes Arrays.sort( ) reusable. The Arrays class doesn't contain any details of how to compare different kinds of objects.

java.lang.Comparable Interface

```
/**
 * Comparable interface defines a lexical
 * ordering for objects in a class.
 */
interface Comparable {
    public int compareTo( Object other );
}
```

a.compareTo(b) < 0	"a comes before b"
a.compareTo(b) = 0	"a and b have same precedence"
a.compareTo(b) > 0	"a comes after b"

Arrays.sort uses Comparable

```
public static void Arrays.sort( Object[] array )
```

The parameter is declared as **`Object[] array`**,
but actually the objects must implement **`Comparable`**.
Otherwise, Arrays.sort will throw an exception.

- Arrays.sort doesn't know (or care) what class of object it will sort.
- Arrays.sort only cares about the *behavior* of the objects in the array:
  the objects must have a compareTo( ) method that defines a lexical order.

Interface with Type Parameter

- Java has type parameters, which make it easier to write typesafe code. In this case <T> represents a datatype:

```
/* Comparable interface with type parameter T.
 * This ensures that you only compare objects
 * of the same type,
 * e.g. string.compareTo(string)
 */
interface Comparable<T> {
   public int compareTo( T other );
}
```

Generics and type parameters were introduced in Java 5.

Example using Type Parameter

"class Student implements Comparable<Student>" means that "T" must be replaced by "Student".

```
public class Student
          implements Comparable<Student> {

     public int compareTo( Student other ) {
        // code for ordering students
      }
```

Implementing Comparable

- To order Students by their ID number we can write:

```
class Student implements Comparable<Student> {
    private String studentId;

    // compare students by ID
    public int compareTo( Student other ) {
        // this code uses the String compareTo
        return
      this.studentId.compareTo(other.studentId);
    }
}
```

This works because studentId is a String and String has compareTo().

Exercise

- What if studentId is a `long`. How would you write compareTo?

```
class Student implements Comparable<Student> {
    private long studentId;
    // compare students by ID
    public int compareTo( Student other ) {
     if (other == null) -1;
     return (int)Math.signum(
            this.studentId - other.studentId);

    }
}
```

Implementing an Interface (C#)

- To declare that your class implements an interface, use:

```
class Student : IComparable {
   private string studentId;
   public int CompareTo( object other ) {
      // compare students by ID
      if ( ! ( other is Student ) )
    throw new Exception("invalid argument");
      // cast as student and compare
      Student s = (Student)other;
      return
         this.studentId.compareTo(s.studentId);
   }
}
```

compareTo consistent with equals

compareTo( ) should be *consistent* with equals( ).

if `a.equals(b)` is **true** then `a.compareTo(b)` == 0

However,

`a.compareTo(b)` == 0 *does not imply* `a.equals(b)` is true.

UML for *Comparable*

Using an external comparator

There are two problems...

1. What if a class does not have a `compareTo` ?

2. What if `compareTo` doesn't do what we want?

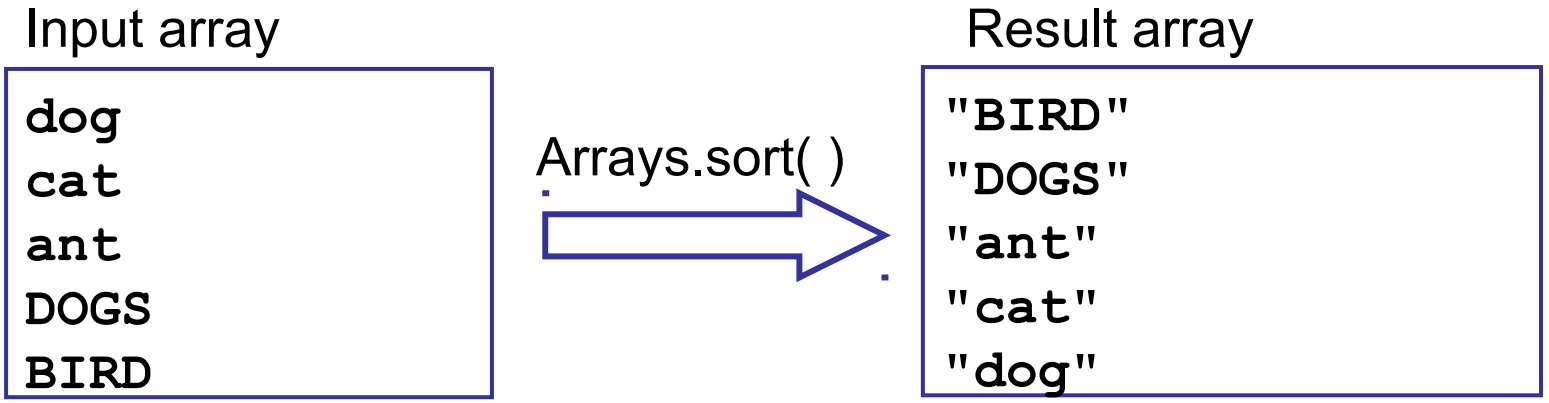
For example...

Sort Strings *ignoring case*

The String compareTo() uses Unicode collation order:

"A" < "Z" < "a" < "b" ...

so, "Bird" comes before "ant".



How can we sort words like in the dictionary (ignore case)?

sort using a Comparator

```
public static void
     Arrays.sort( T[] array, Comparator<T> c )
```


This sort( ) method uses an external *Comparator* object to compare values in the array.

So, what is a *Comparator?*

Can you guess?

java.util.Comparator Interface

```
/**
 * A Comparator defines an ordering of
 * objects of same class (or class
 * hierarchy).
 */
interface Comparator<T> {
    public int compare( T a, T b );
}
```

compare(a, b) < 0 "a comes before b"

compare(a, b) = 0 "a and b have same precedence"

compare(a, b) > 0 "a comes after b"

In Java 8, *Comparator* has many more methods, but you can ignore them. Only compare(a,b) is required.

Implementing Comparator

- Order Strings ignoring case

```
class CompareIgnoreCase
          implements Comparator<String> {

   public int compare(String a, String b) {
        return a.compareToIgnoreCase(b);
        //TODO check that a and b are not null.
   }
```

Another Example

- Order Students by ID. If the ID is same, then order by name.

```
class CompareById
          implements Comparator<Student> {

   public int compare(Student a, Student b) {
      int comp =
            Long.compare(a.getId(), b.getId());
      // if ID is same then order by name
      if (comp == 0) comp =
            a.getName().compareTo(b.getName());
      return comp;
   }
```

Exercise: Write a Comparator

Write a Comparator that order strings by length, shortest length first. If length is same, order alphabetically.

```
String[] words =
        {"ants","cat","dog","Elephant","zebra"};
Comparator<String> byLength =
                          new CompareByLength();
Arrays.sort( words, byLength );
```

```
words[0] = "cat"
words[1] = "dog"
words[2] = "ants"
words[3] = "zebra"
words[4] = "Elephant"
```

Sorting a List

You can sort Lists the same way as arrays.

The methods are:

Collections.sort(List list) - sort a list using compareTo.

Collections.sort(List<T> list, Comparator<T> cmp)

- sort a list using an external *Comparator*.

Collections is in java.util. You should study it.

Review

What are 3 methods for sorting an array?

What interfaces have you studied so far in OOP?