Interfaces - Recap

CPSC 1181 – O.O.

Jeremy Hilliker

Summer 2017

Langara.

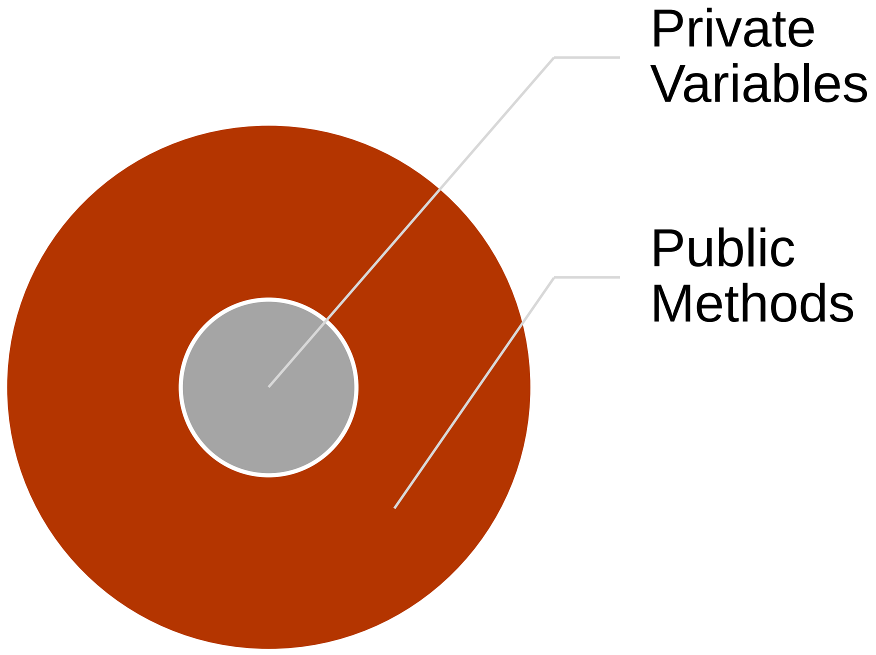
THE COLLEGE OF HIGHER LEARNING.

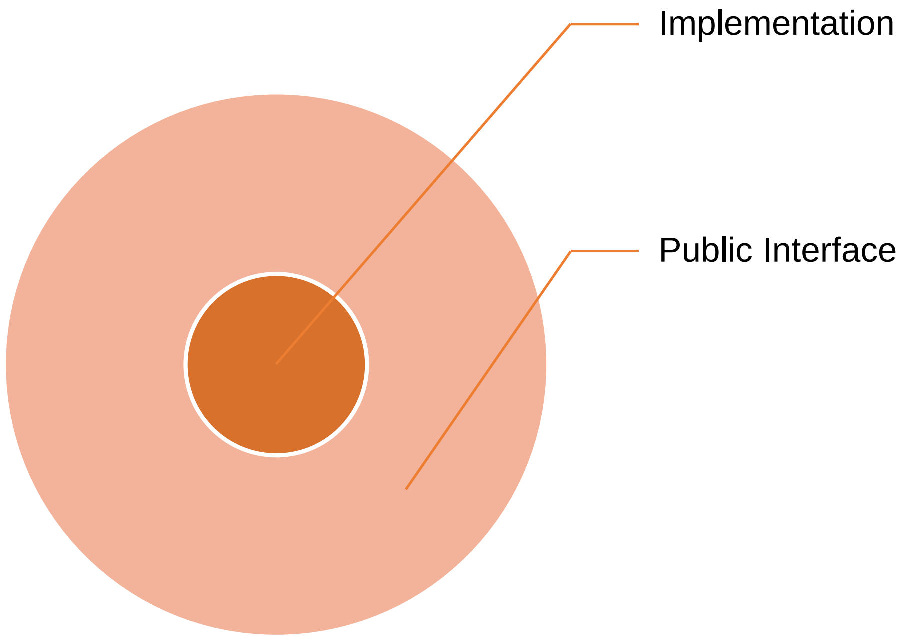
Overview

- Review
- Examples

Recall:

- Encapsulation
- Abstraction





What’s an Interface?

- A surface of the Encapsulation and Abstraction shells
  - A set of methods with no* implementation
  - And no instance variables
- A set of behaviours
  - Values (state) are implicit through the behaviours
- A type
  - Substitution (assignment)
  - Polymorphism
- A “contract”
  - If this, then this
  - “adds the given element to the end of the list”
  - “returns the position of the element in the list, or -1 if not present”

What’s an Interface?

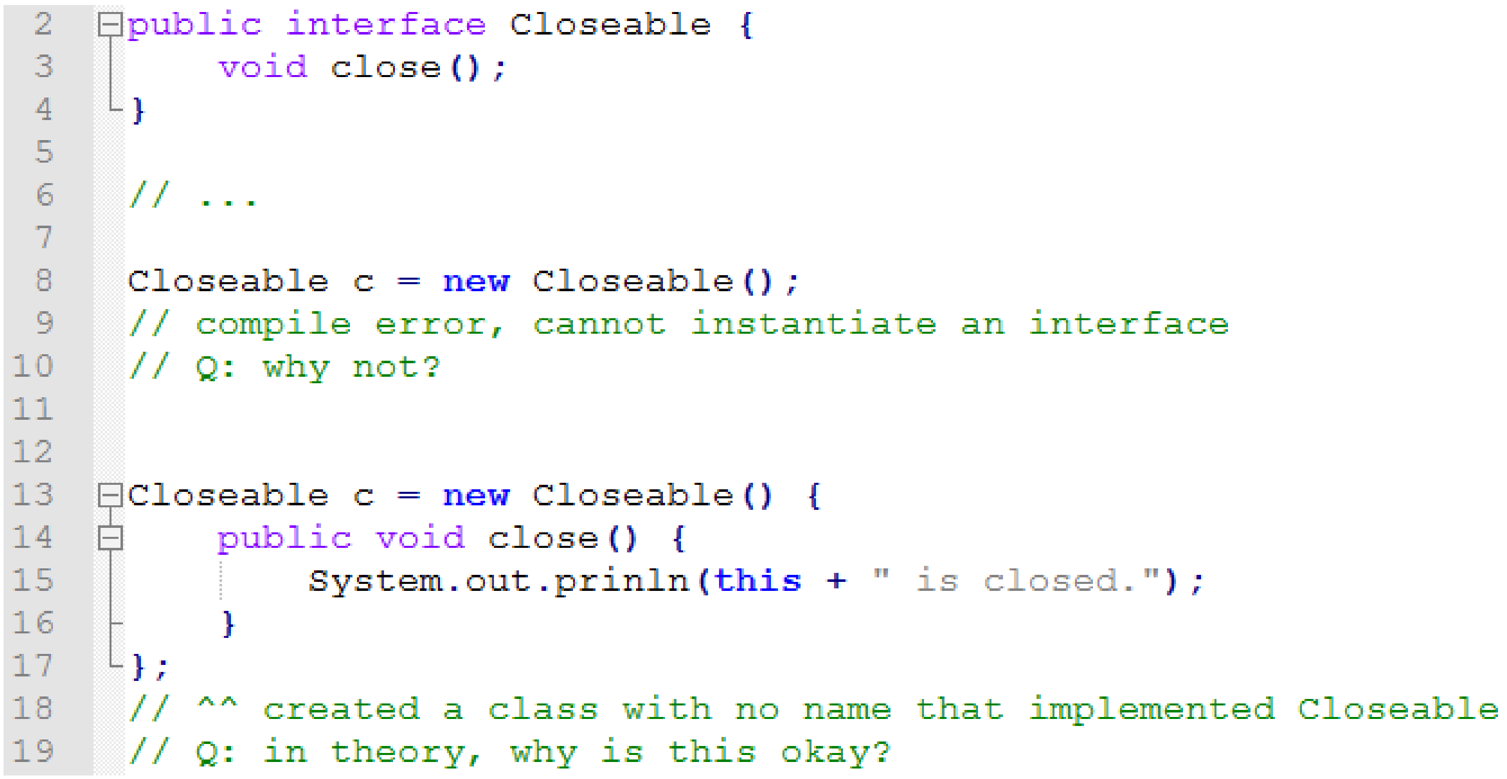
- A reference ***type***
- A set of *abstract methods and constants*
  - And nested types (inner interfaces)
  - In Java 8:
    - “default” methods (boo)
    - static methods (boo)
- A class implements an interface
  - Inheriting the abstract methods (and constants)

Q: can you instantiate an interface?

- No
- An interface is a reference *type*
  - Assignable -> substitution
- It's *abstract*, not a concrete type
- It's not a class -> cannot create objects

- Class: template for creating objects of a type
- Type: set of values and the operations on those values
- Interface:
  - collection of abstract methods and constants
    - No implementation
  - Values (state) are implicit through behaviours

But… Anonymous Class

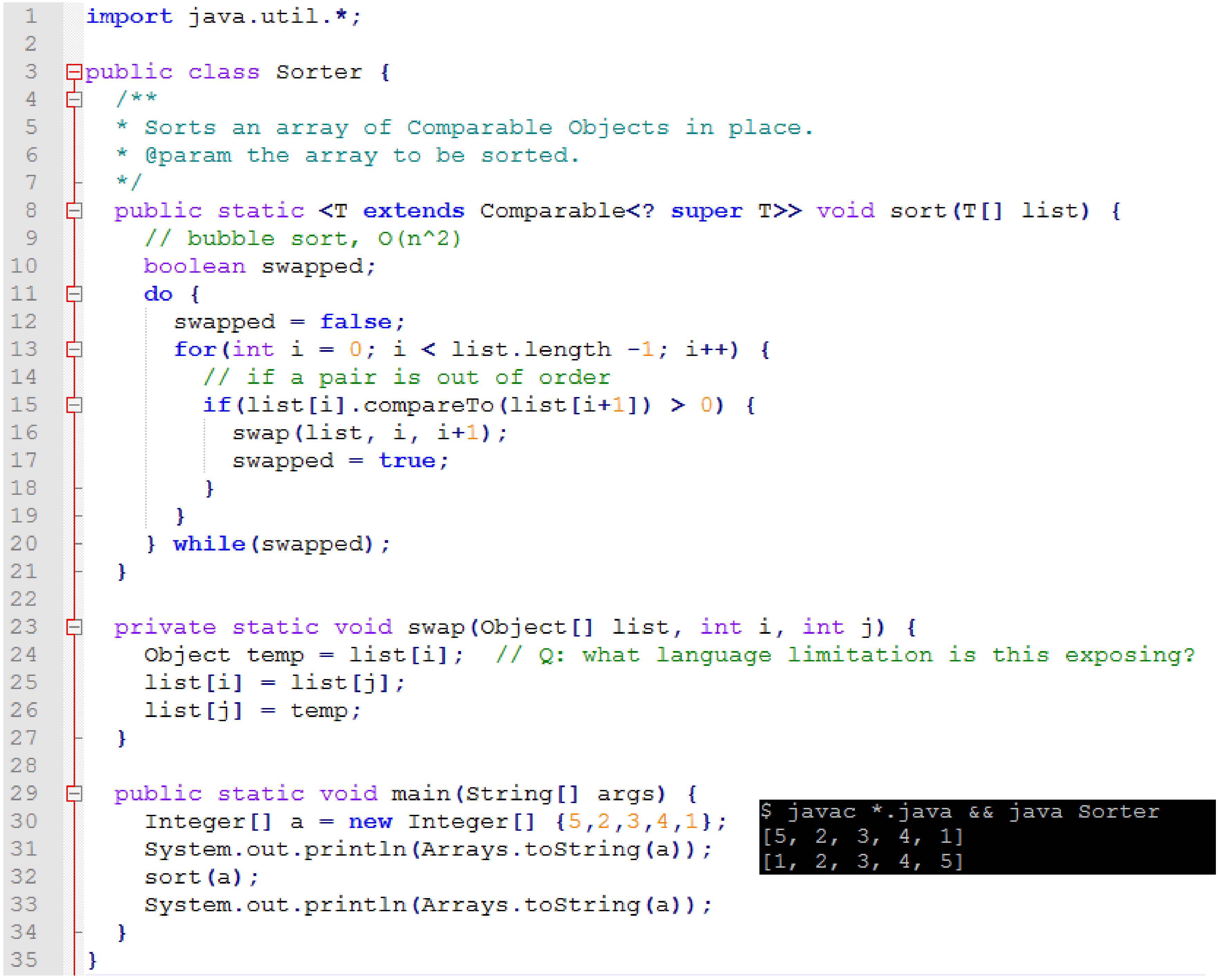


```
public interface Closeable {
    void close();
}

// ...

Closeable c = new Closeable();
// compile error, cannot instantiate an interface
// Q: why not?


Closeable c = new Closeable() {
    public void close() {
        System.out.prinln(this + " is closed.");
    }
};
// ^^ created a class with no name that implemented Closeable
// Q: in theory, why is this okay?
```

Interface vs Class

- Similar
  - Contains any number of methods (inc. 0)
  - Written in a .java file, names must match
  - Bytecode compiles to a .class file
  - Appear in packages with same rules as classes
- Different
  - Cannot be instantiated (abstract)
  - No constructors
  - All methods are abstract
  - No instance variables, on final static
  - Class implements interface, not extends
  - Class may implement multiple interfaces
  - Interface may extend multiple interfaces

Sorting



```java
import java.util.*;

public class Sorter {
  /**
  * Sorts an array of Comparable Objects in place.
  * @param the array to be sorted.
  */
  public static <T extends Comparable<? super T>> void sort(T[] list) {
    // bubble sort, O(n^2)
    boolean swapped;
    do {
      swapped = false;
      for(int i = 0; i < list.length -1; i++) {
        // if a pair is out of order
        if(list[i].compareTo(list[i+1]) > 0) {
          swap(list, i, i+1);
          swapped = true;
        }
      }
    } while(swapped);
  }

  private static void swap(Object[] list, int i, int j) {
    Object temp = list[i];  // Q: what language limitation is this exposing?
    list[i] = list[j];
    list[j] = temp;
  }

  public static void main(String[] args) {
    Integer[] a = new Integer[] {5,2,3,4,1};
    System.out.println(Arrays.toString(a));
    sort(a);
    System.out.println(Arrays.toString(a));
  }
}
```

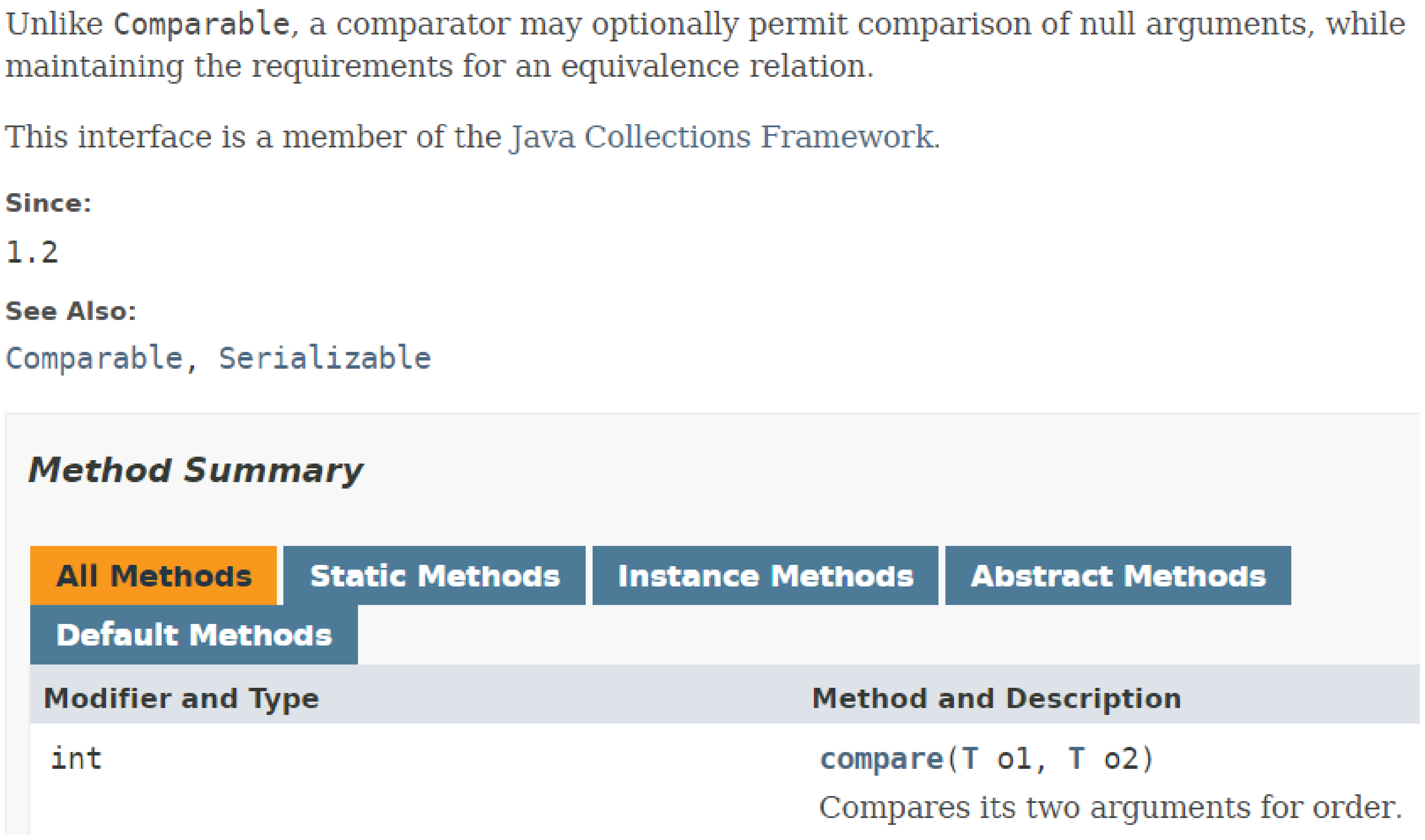
```
$ javac *.java && java Sorter
[5, 2, 3, 4, 1]
[1, 2, 3, 4, 5]
```

Sorting

- Used “is-a” Comparable
- What if we want to sort things that are not “is-a” Comparable?

1. Extend them
    - What if there are multiple things in the hierarchy?
        - Have to extend each one at every level
    - What if they return things that we want to sort too
        - Have to extend those and wrap them on each call
    - What if we want to change what attributes to compare for sorting at runtime?
    - This is all a huge pain
2. Make a class that compares them
    - We will specify its interface

Interface java.util.Comparator<T>



Unlike `Comparable`, a comparator may optionally permit comparison of null arguments, while maintaining the requirements for an equivalence relation.

This interface is a member of the Java Collections Framework.

Since:

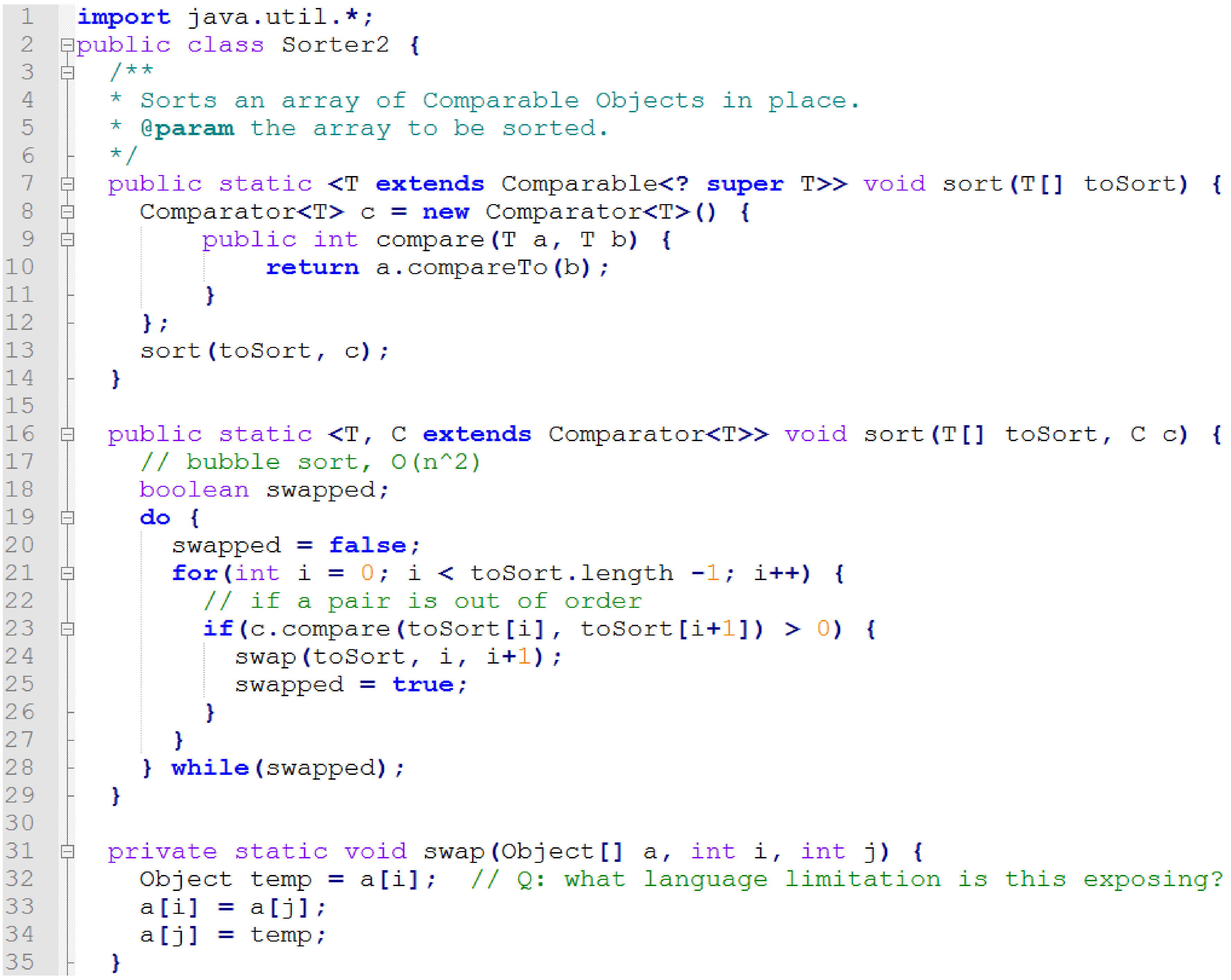
1.2

See Also:

`Comparable`, `Serializable`

Method Summary

Modifier and Type	Method and Description
`int`	`compare(T o1, T o2)` Compares its two arguments for order.

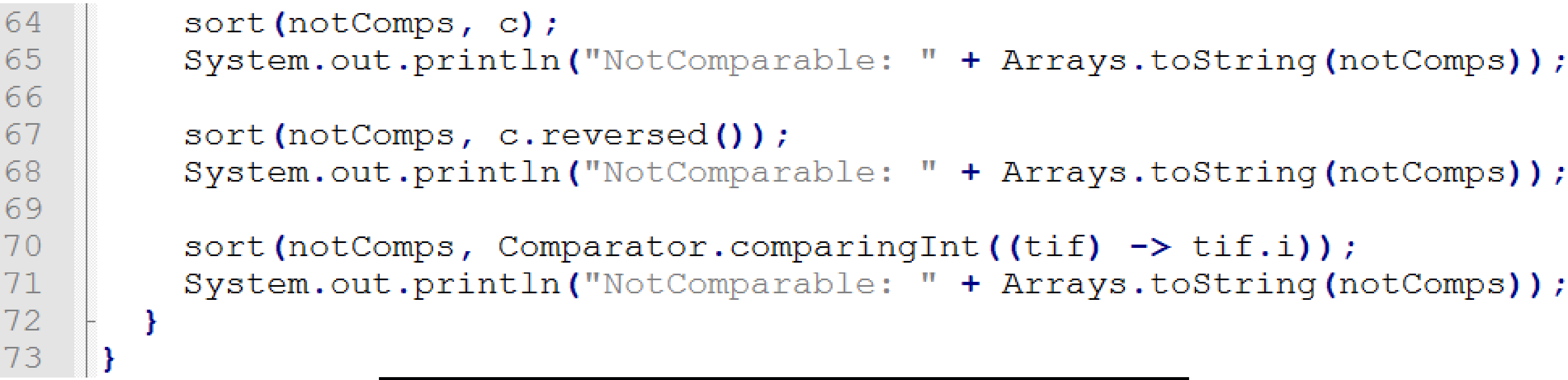


```java
import java.util.*;
public class Sorter2 {
  /**
  * Sorts an array of Comparable Objects in place.
  * @param the array to be sorted.
  */
  public static <T extends Comparable<? super T>> void sort(T[] toSort) {
    Comparator<T> c = new Comparator<T>() {
        public int compare(T a, T b) {
            return a.compareTo(b);
        }
    };
    sort(toSort, c);
  }

  public static <T, C extends Comparator<T>> void sort(T[] toSort, C c) {
    // bubble sort, O(n^2)
    boolean swapped;
    do {
      swapped = false;
      for(int i = 0; i < toSort.length -1; i++) {
        // if a pair is out of order
        if(c.compare(toSort[i], toSort[i+1]) > 0) {
          swap(toSort, i, i+1);
          swapped = true;
        }
      }
    } while(swapped);
  }

  private static void swap(Object[] a, int i, int j) {
    Object temp = a[i];  // Q: what language limitation is this exposing?
    a[i] = a[j];
    a[j] = temp;
  }
```

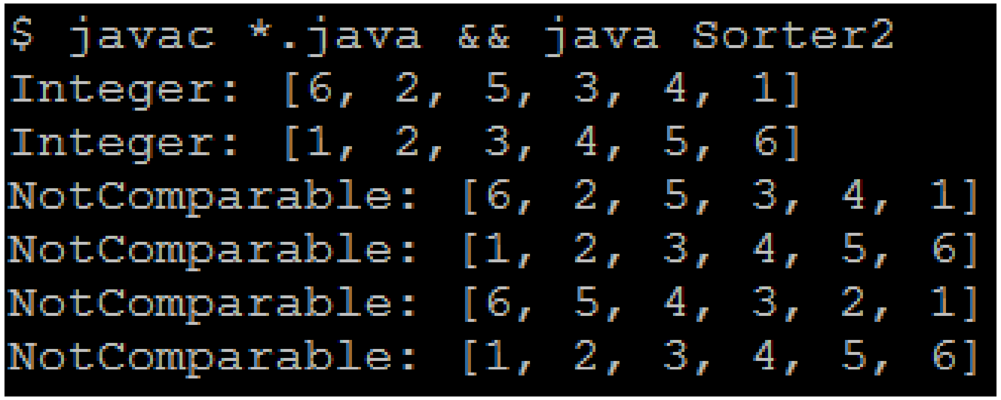
```java
public static void main(String[] args) {
    final Integer[] values = new Integer[] {6,2,5,3,4,1};

    Integer[] a = values.clone();
    System.out.println("Integer: " + Arrays.toString(a));
    sort(a);
    System.out.println("Integer: " + Arrays.toString(a));

    class NotComparable {
        public final int i;
        public NotComparable(int i) { this.i = i; }
        public String toString() { return Integer.toString(i); }
    }
    NotComparable[] notComps = new NotComparable[values.length];
    for(int i = 0; i < notComps.length; i++) {
        notComps[i] = new NotComparable(values[i]);
    }

    System.out.println("NotComparable: " + Arrays.toString(notComps));
    //sort(notComps); // compile error, notComps is-not-a Comparable

    Comparator<NotComparable> c = new Comparator<NotComparable>() {
        public int compare(NotComparable a, NotComparable b) {
            return a.i - b.i;
        }
    };

    sort(notComps, c);
    System.out.println("NotComparable: " + Arrays.toString(notComps));
```

```
$ javac *.java && java Sorter2
Integer: [6, 2, 5, 3, 4, 1]
Integer: [1, 2, 3, 4, 5, 6]
NotComparable: [6, 2, 5, 3, 4, 1]
NotComparable: [1, 2, 3, 4, 5, 6]
```

Other Affordances of Comparator



```java
        sort(notComps, c);
        System.out.println("NotComparable: " + Arrays.toString(notComps));

        sort(notComps, c.reversed());
        System.out.println("NotComparable: " + Arrays.toString(notComps));

        sort(notComps, Comparator.comparingInt((tif) -> tif.i));
        System.out.println("NotComparable: " + Arrays.toString(notComps));
    }
}
```



```
$ javac *.java && java Sorter2
Integer: [6, 2, 5, 3, 4, 1]
Integer: [1, 2, 3, 4, 5, 6]
NotComparable: [6, 2, 5, 3, 4, 1]
NotComparable: [1, 2, 3, 4, 5, 6]
NotComparable: [6, 5, 4, 3, 2, 1]
NotComparable: [1, 2, 3, 4, 5, 6]
```

Recap

- Encapsulation & Abstraction
- What is an interface?
  - A *set of abstract methods and constants*
- No instantiation
  - Anonymous classes
- Compare Interface to Class
- Ex:
  - Sorting
    - Comparable
    - Comparator