Interfaces

CPSC 1181 – O.O.

Jeremy Hilliker

Summer 2017

Langara.

THE COLLEGE OF HIGHER LEARNING.

Overview

- Interfaces

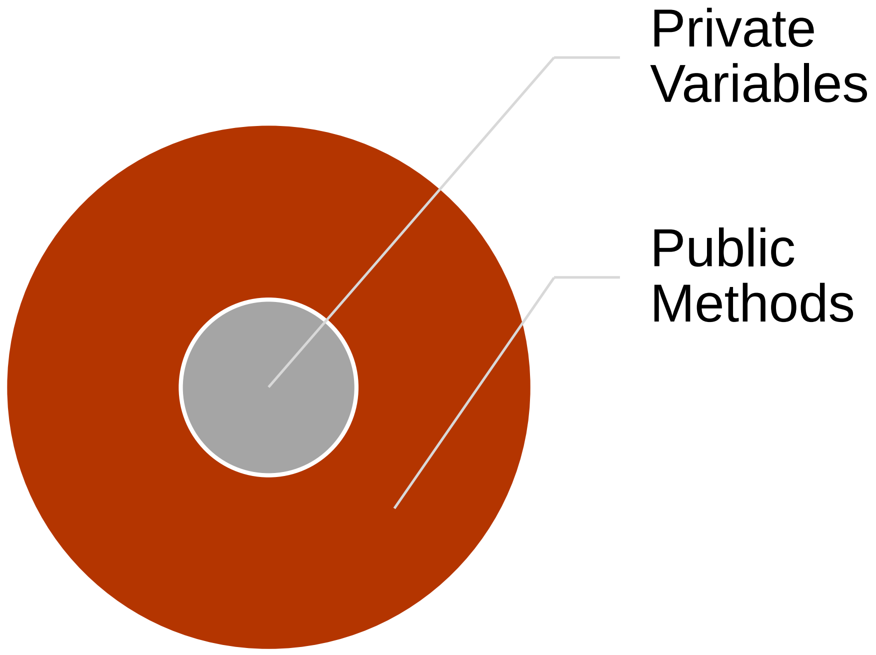
- Constants
- Implementing
- Extending
- Multiple Interfaces
- Code Reuse
- List<E>

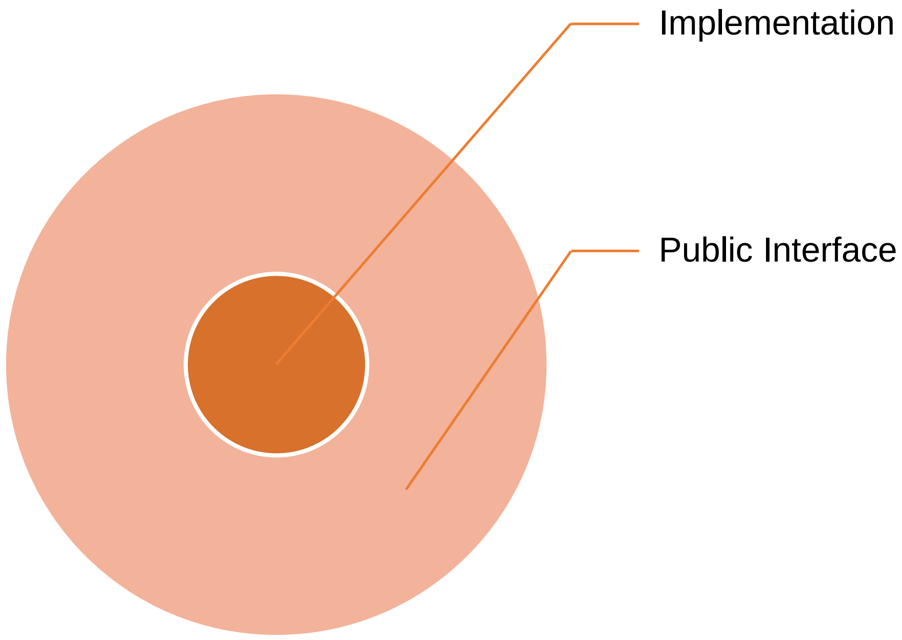
Recall: Three Big Ideas of O.O.

- Encapsulation (data hiding):
  - restricting direct access to some of the object's components (ie: variables)
  - bundling of data with the methods operating on that data (ie: a class)
- Abstraction:
  - Dealing with ideas rather than events
  - Providing functionality
  - Hiding implementation details
  - Know: what it does, not how it does it
  - “Design by Contract”
- Polymorphism
- [Inheritance (and to a lesser extent, composition)]

Encapsulation

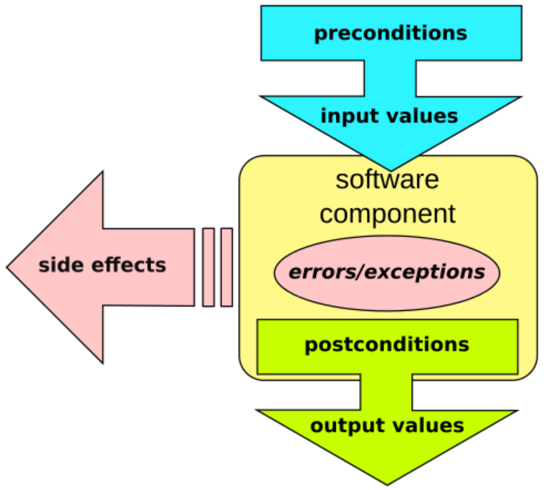
Abstraction





Recall: 2) Public Interface

- Methods through which the object is manipulated
  - Encapsulation:
    - hiding the object's variables (private modifier)
    - Only members of the class may access them directly
  - Abstraction:
    - Hiding the details (the implementation)
  - You can (read: should) have private helper methods



Problem

- We’ve been dealing with classes
  - Have full implementation
  - We know how they are implemented
  - We know what variables are present, though we may not know their values
- Don’t really have Abstraction

Ex:

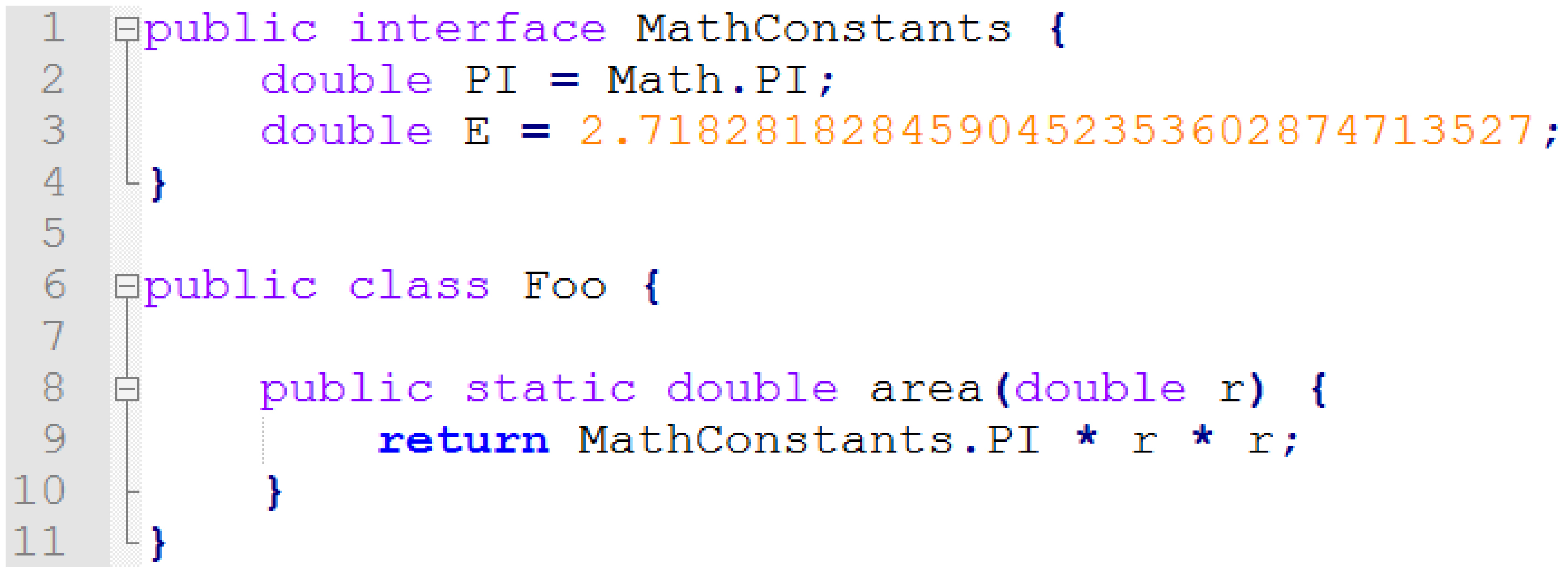
- ArrayList<>
  - From javadoc, I know:
    - Backed by an array
    - Of a fixed size
    - Grows as necessary
      - In a predictable way
    - Probably doesn’t automatically shrink
  - I can find all of the source code for this class if I want to.
- There’s no real abstraction
  - I can start depending on behaviours of ArrayList that are not part of it’s public interface
  - That’s bad

Solution: Interfaces

- Have a language construct that ***only*** specifies the public interface

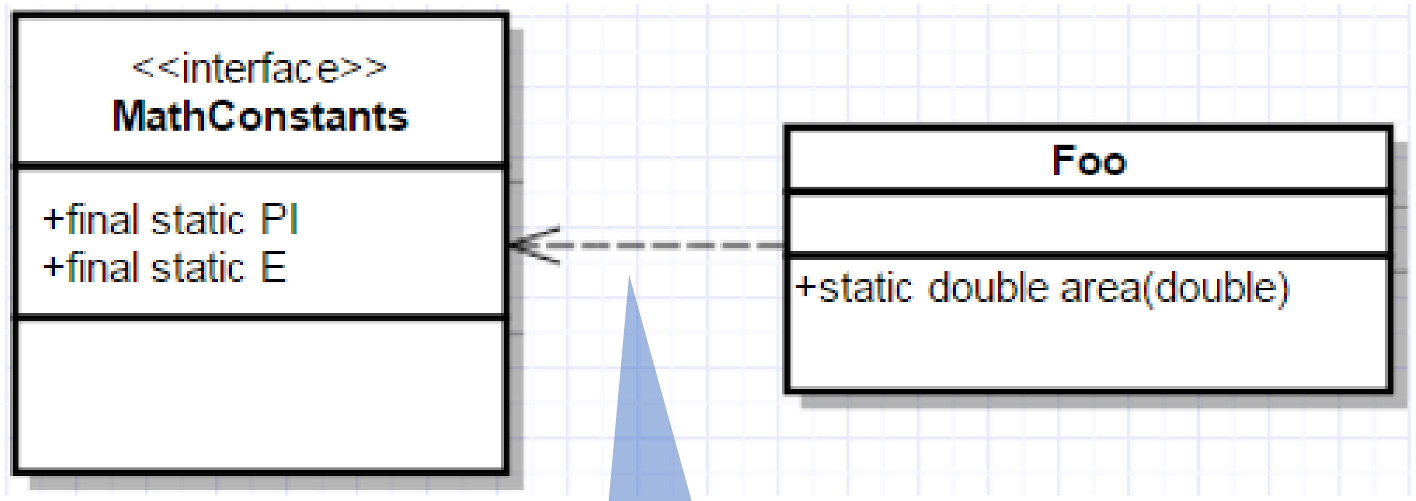
- Only constants, no variables
  - Encapsulation: Data is hidden
  - Abstraction: Cant know of variable's existence
- Only public methods, no other modifiers
  - Encapsulation: things grouped together
  - Abstraction: Cant know any implementation

Using an Interface - Constants



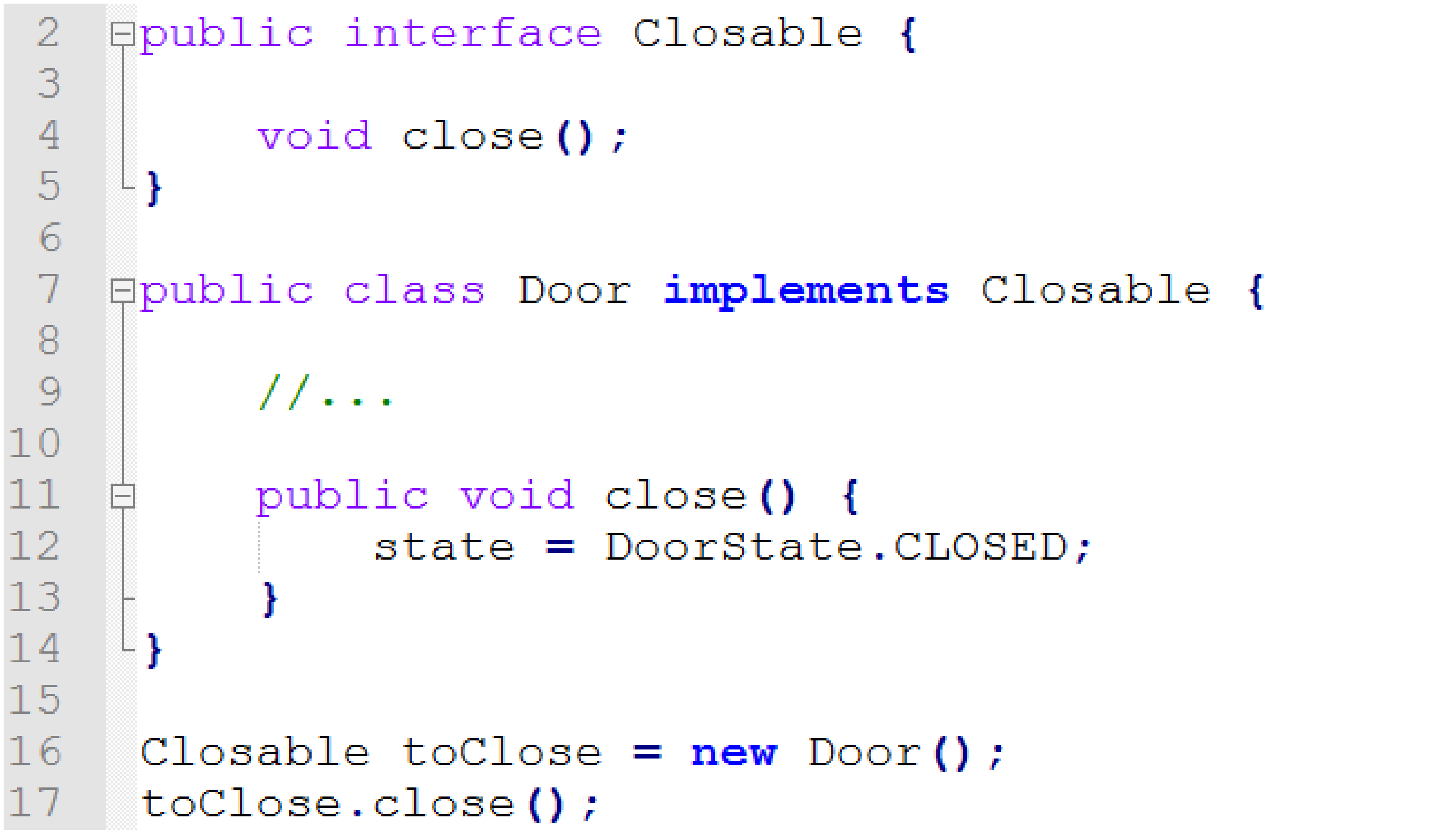
```java
public interface MathConstants {
      double PI = Math.PI;
      double E = 2.7182818284590452353602874713527;
}

public class Foo {

      public static double area(double r) {
            return MathConstants.PI * r * r;
      }
}
```

Using an Interface - Constants



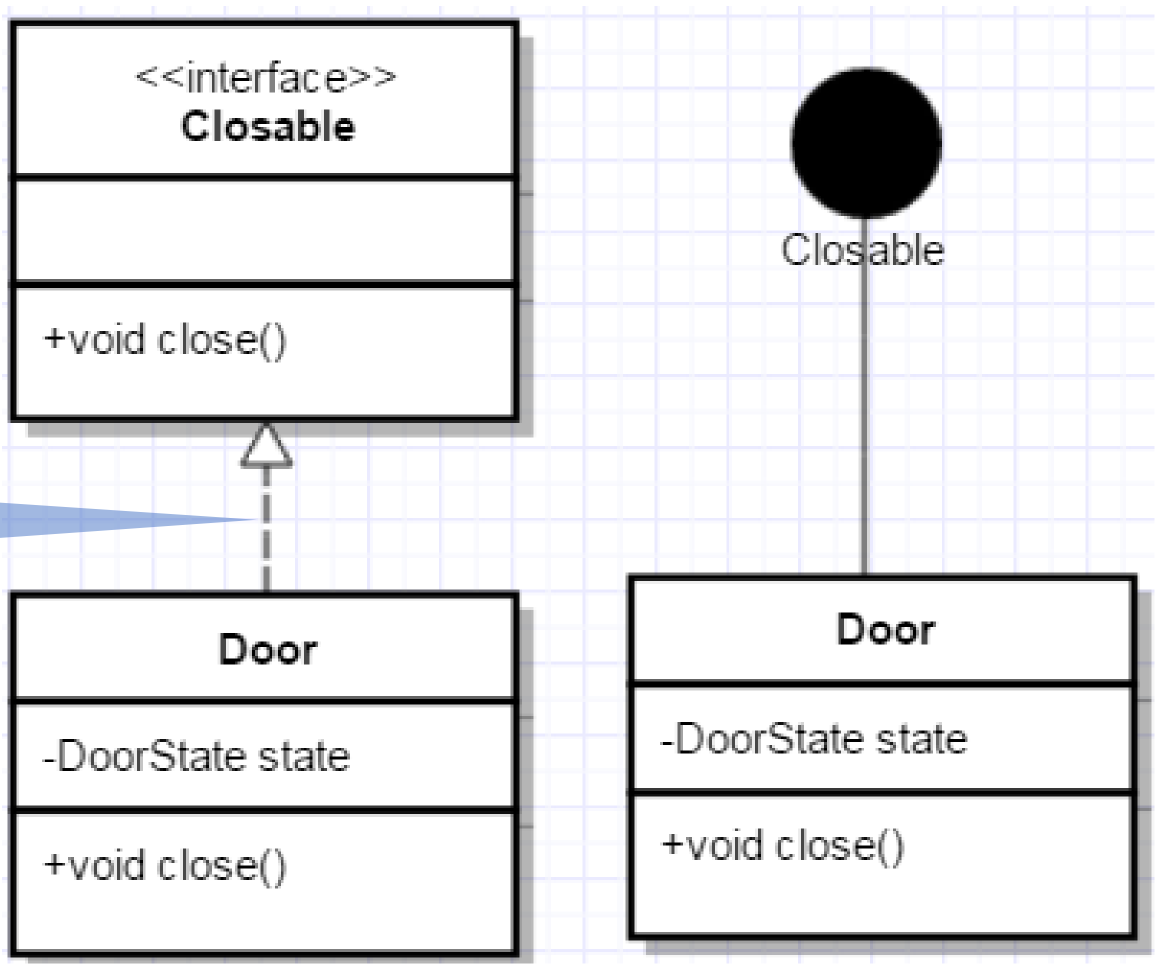
Dependency / “uses” arrow

Implementing - Closable



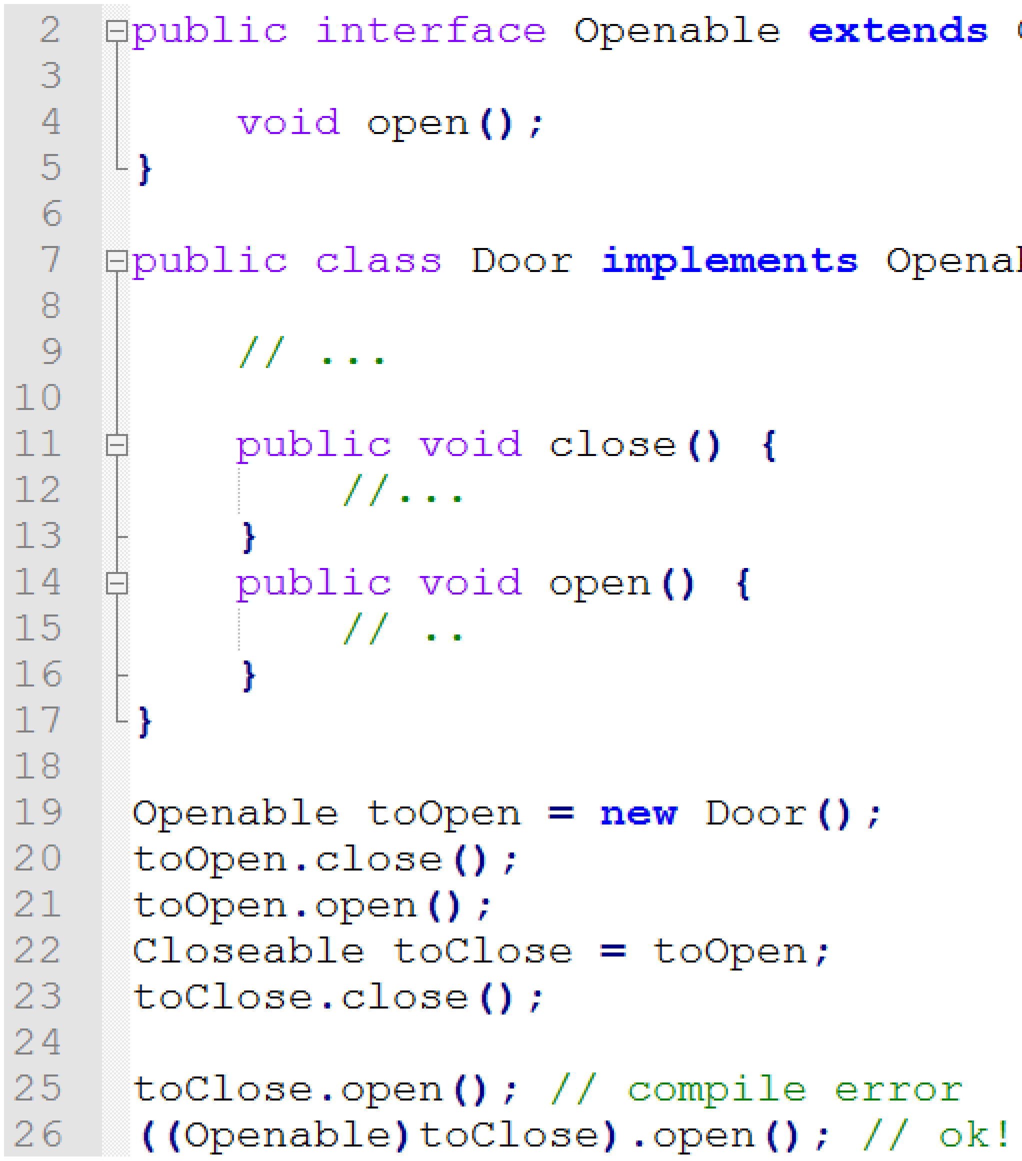
```java
public interface Closable {

    void close();
}

public class Door implements Closable {

    //...

    public void close() {
        state = DoorState.CLOSED;
    }
}

Closable toClose = new Door();
toClose.close();
```

Implementing - Closable



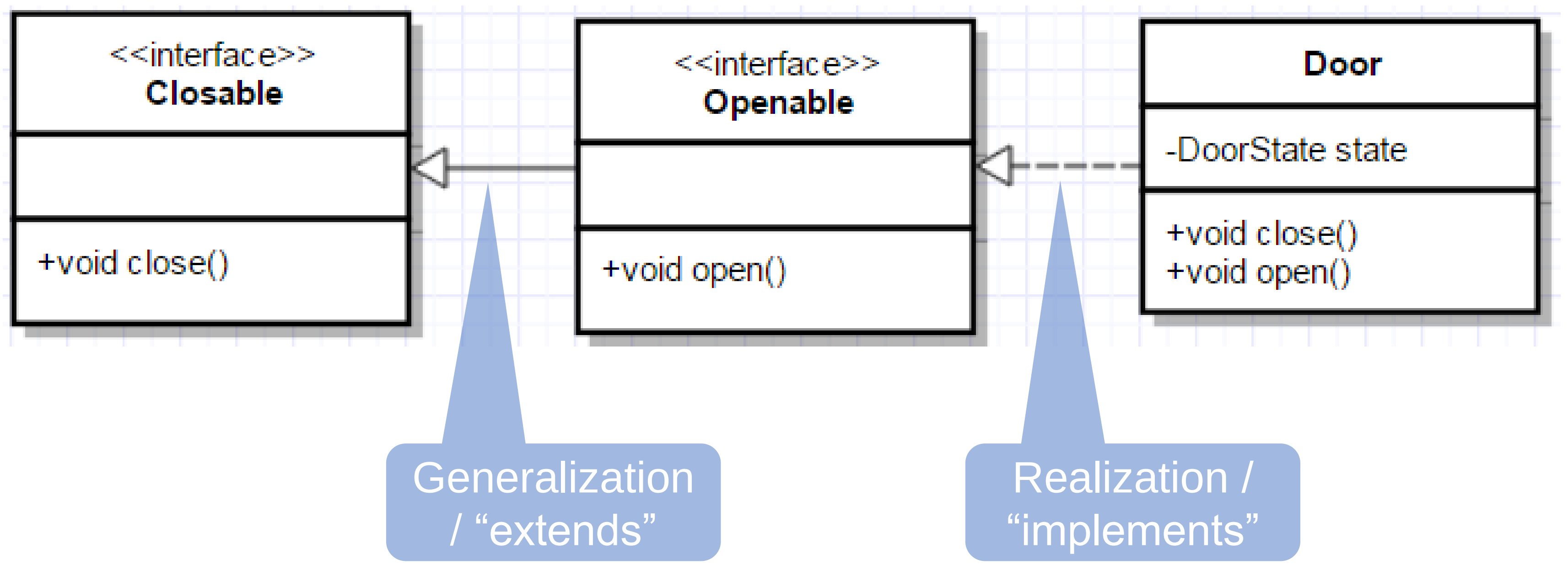
Realization /
“Implements”

Extending - Openable

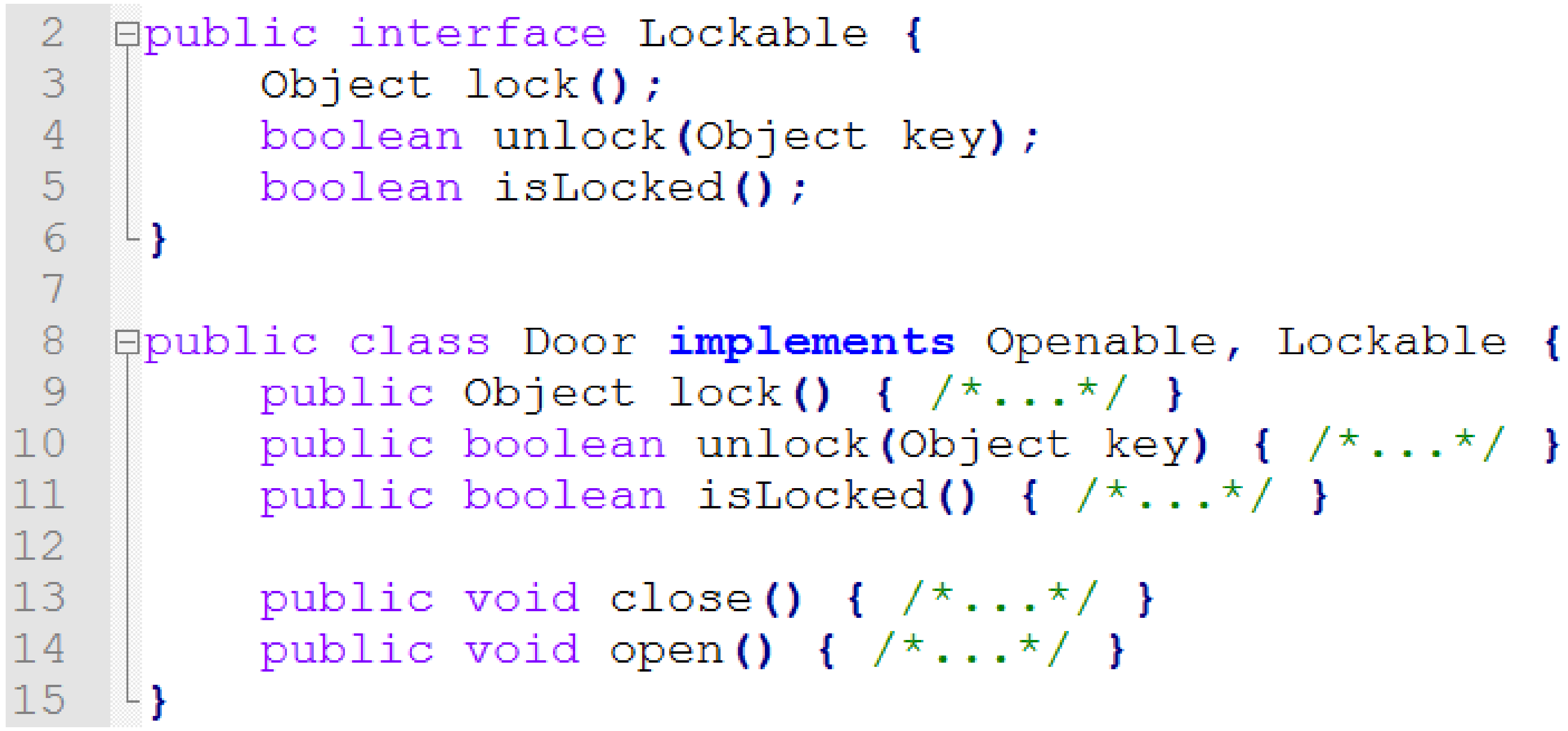


```java
public interface Openable extends Closeable {

    void open();
}

public class Door implements Openable {

    // ...

    public void close() {
        //...
    }
    public void open() {
        // ..
    }
}

Openable toOpen = new Door();
toOpen.close();
toOpen.open();
Closeable toClose = toOpen;
toClose.close();

toClose.open(); // compile error
((Openable)toClose).open(); // ok!
```

Extending - Openable

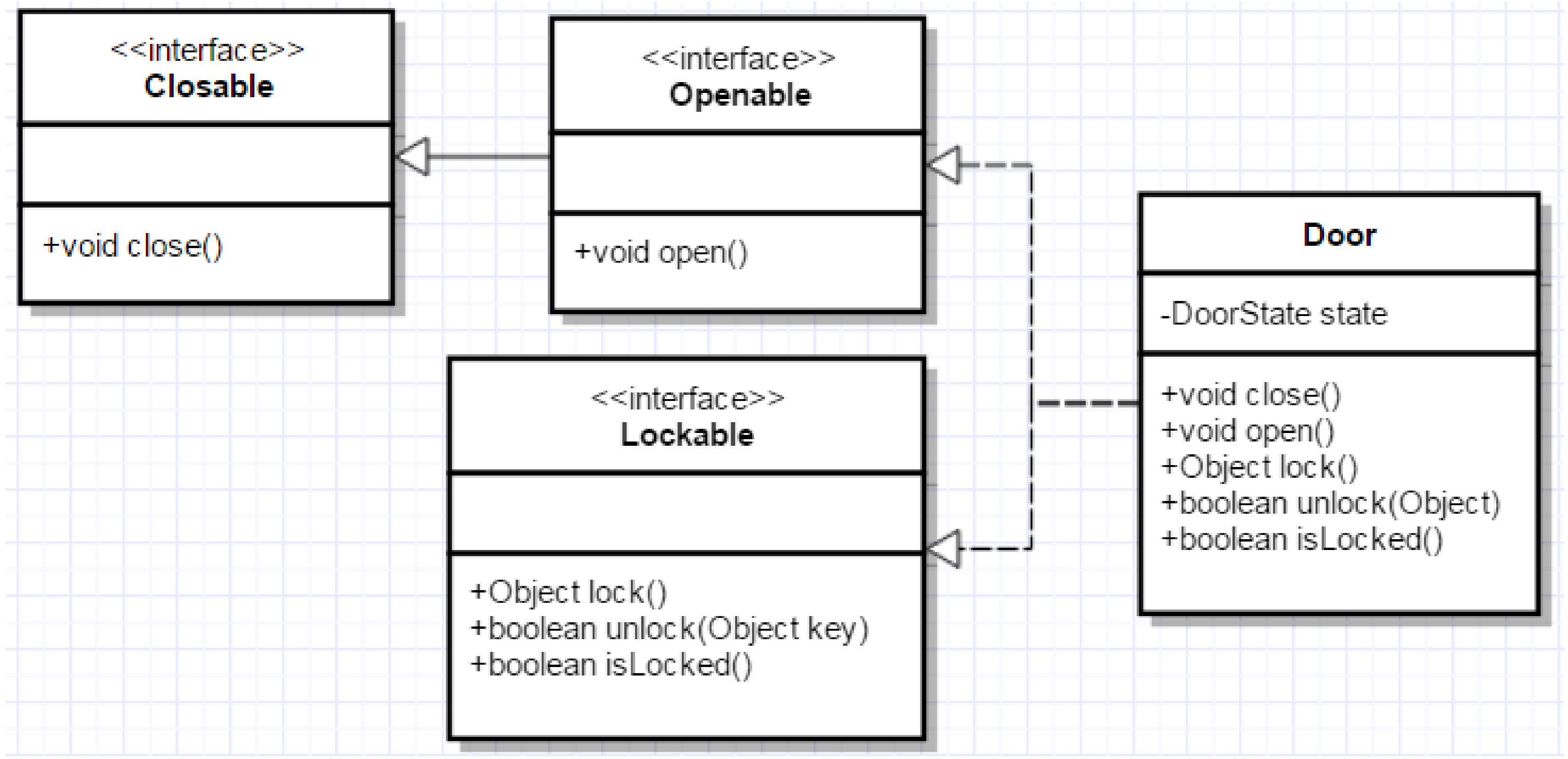


Multiple Interfaces - Openable & Lockable



```java
public interface Lockable {
    Object lock();
    boolean unlock(Object key);
    boolean isLocked();
}

public class Door implements Openable, Lockable {
    public Object lock() { /*...*/ }
    public boolean unlock(Object key) { /*...*/ }
    public boolean isLocked() { /*...*/ }

    public void close() { /*...*/ }
    public void open() { /*...*/ }
}
```

Multiple Interfaces - Openable & Lockable

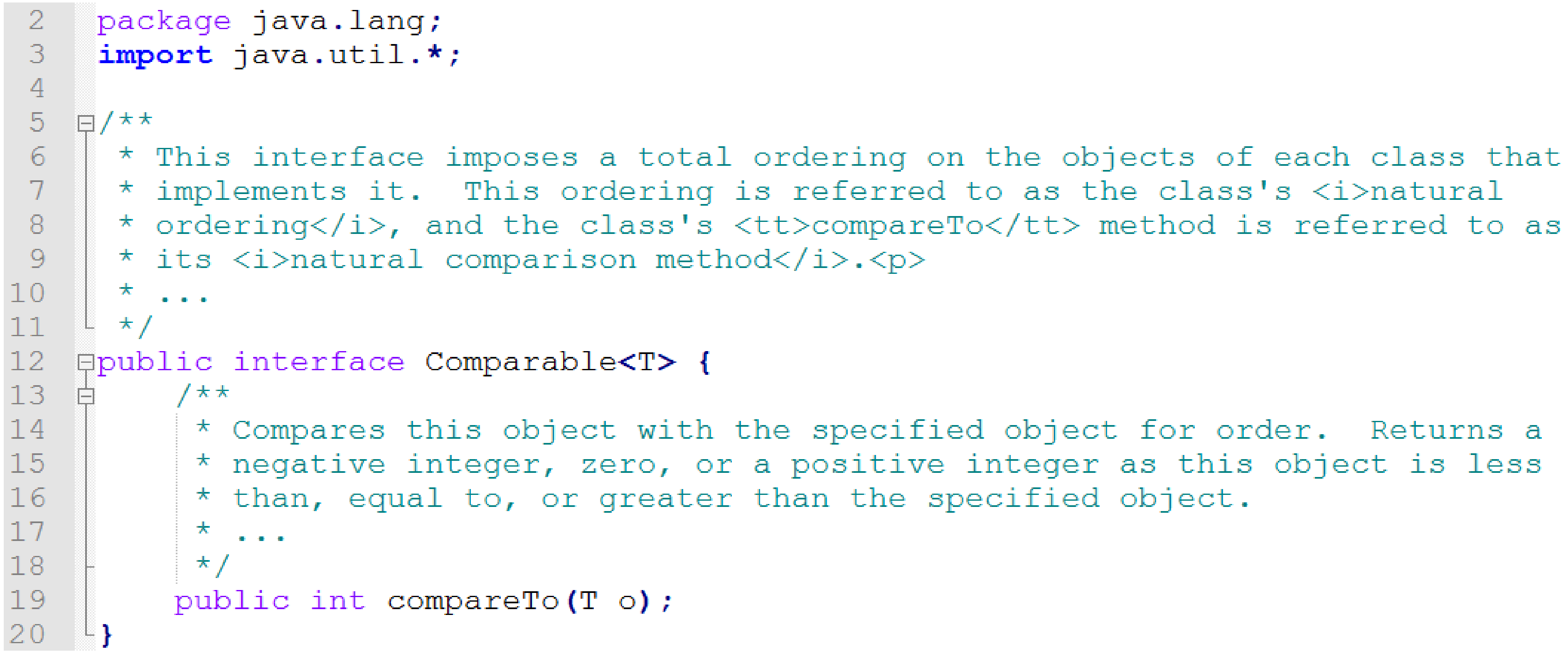


Why?

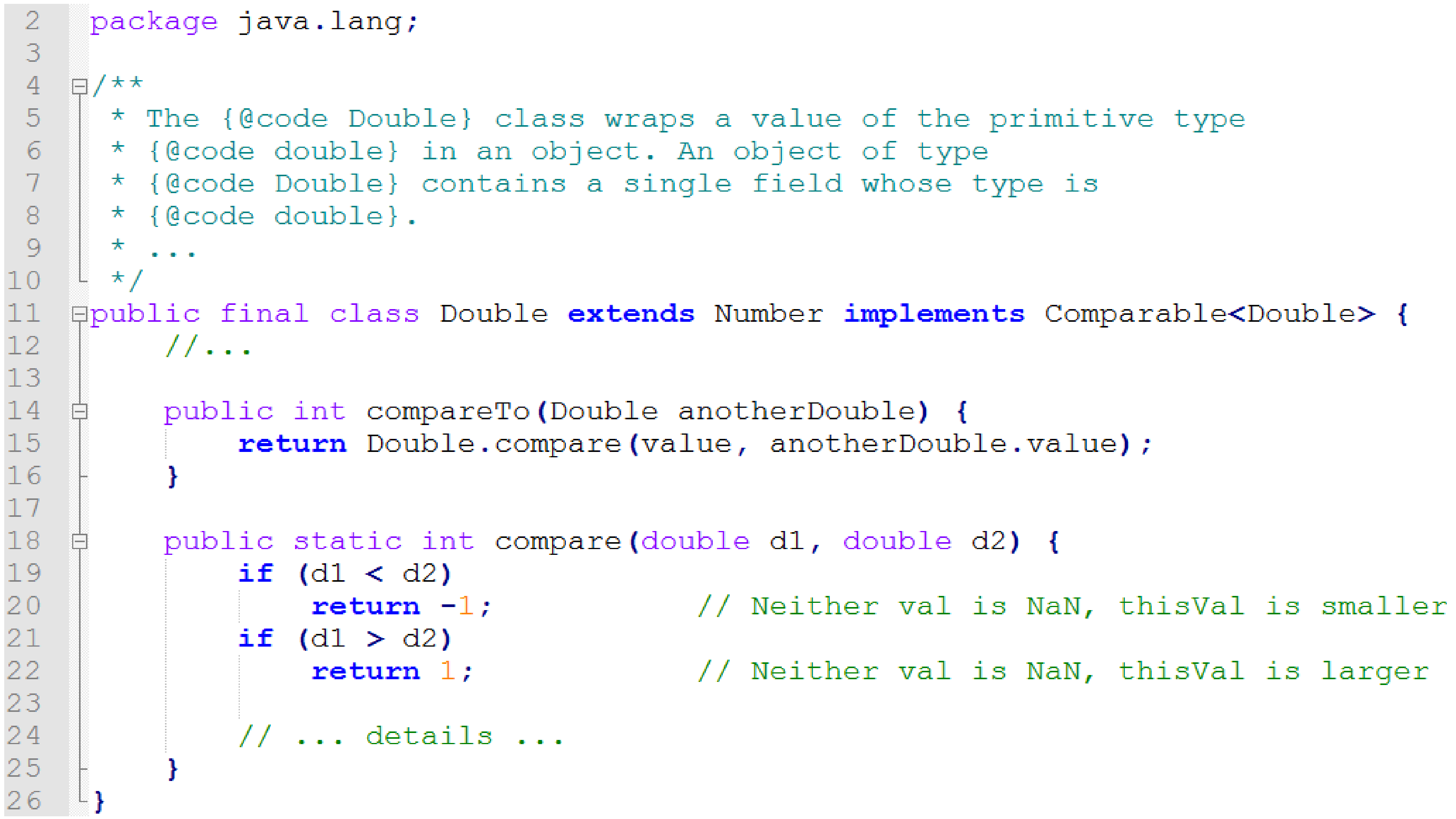
- Okay in theory, but why in practice?

- For code reuse.

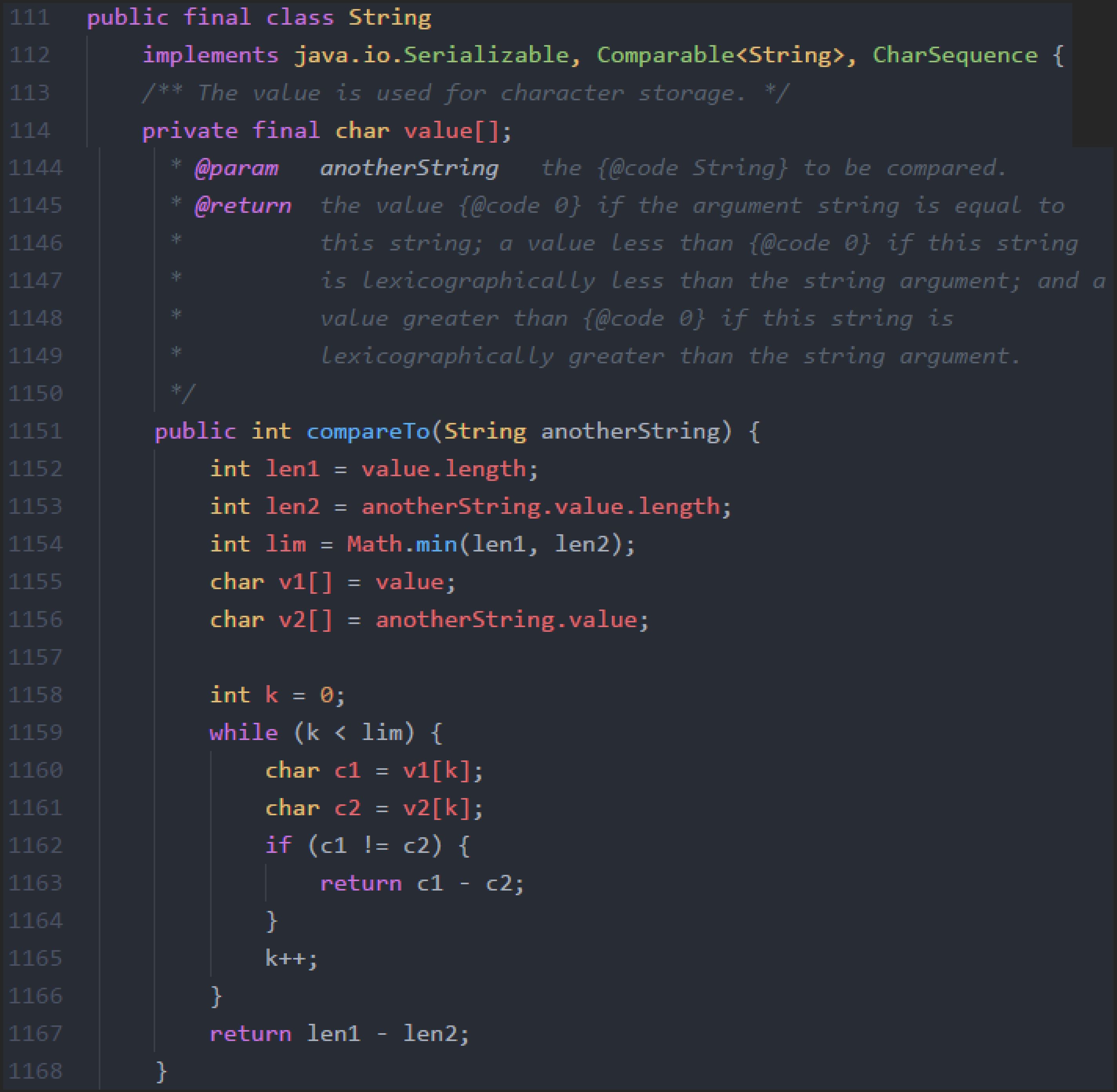
Code Reuse - Comparable



```java
package java.lang;
import java.util.*;

/**
 * This interface imposes a total ordering on the objects of each class that
 * implements it.  This ordering is referred to as the class's <i>natural
 * ordering</i>, and the class's <tt>compareTo</tt> method is referred to as
 * its <i>natural comparison method</i>.<p>
 * ...
 */
public interface Comparable<T> {
    /**
     * 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.
     * ...
     */
    public int compareTo(T o);
}
```



```java
package java.lang;

/**
 * The {@code Double} class wraps a value of the primitive type
 * {@code double} in an object. An object of type
 * {@code Double} contains a single field whose type is
 * {@code double}.
 * ...
 */
public final class Double extends Number implements Comparable<Double> {
    //...

    public int compareTo(Double anotherDouble) {
        return Double.compare(value, anotherDouble.value);
    }

    public static int compare(double d1, double d2) {
        if (d1 < d2)
            return -1;           // Neither val is NaN, thisVal is smaller
        if (d1 > d2)
            return 1;            // Neither val is NaN, thisVal is larger

        // ... details ...
    }
}
```



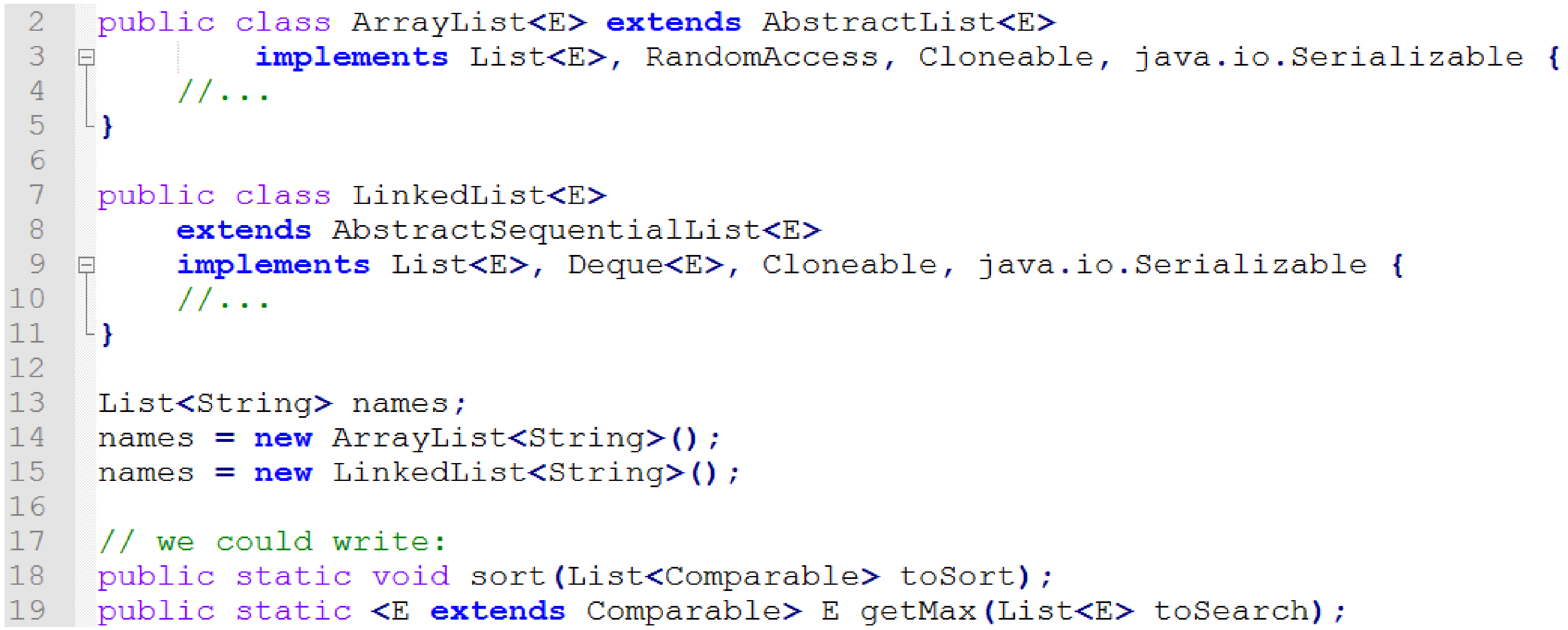
```java
public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {
    /** The value is used for character storage. */
    private final char value[];
     * @param   anotherString   the {@code String} to be compared.
     * @return  the value {@code 0} if the argument string is equal to
     *          this string; a value less than {@code 0} if this string
     *          is lexicographically less than the string argument; and a
     *          value greater than {@code 0} if this string is
     *          lexicographically greater than the string argument.
     */
    public int compareTo(String anotherString) {
        int len1 = value.length;
        int len2 = anotherString.value.length;
        int lim = Math.min(len1, len2);
        char v1[] = value;
        char v2[] = anotherString.value;

        int k = 0;
        while (k < lim) {
            char c1 = v1[k];
            char c2 = v2[k];
            if (c1 != c2) {
                return c1 - c2;
            }
            k++;
        }
        return len1 - len2;
    }
```

Why?

- We can write methods that can accept anything that is-a Comparable
  - Including things that we never thought about
- If anyone wants to use our methods
  - They just implement Comparable

Eg: List<>

- ArrayList



```java
public class ArrayList<E> extends AbstractList<E>
        implements List<E>, RandomAccess, Cloneable, java.io.Serializable {
    //...
}

public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable {
    //...
}

List<String> names;
names = new ArrayList<String>();
names = new LinkedList<String>();

// we could write:
public static void sort(List<Comparable> toSort);
public static <E extends Comparable> E getMax(List<E> toSearch);
```

List<>

- 2 classes that implement List<>
  - Each “is a” List<>
    - Can do all “list” things
    - Can be assigned to a List<> variable
  - Have their own implementations

- When we get a List<>,
  - cant get any instance variables (encapsulation)
    - Don’t know which implementation
  - no idea of the implementation (abstraction)
    - Could even be another kind of list entirely
      - Eg: Collections.unmodifiableList(List<>)

List<>

- ArrayList<E> is-a:
  - AbstractList<E> (extends)
  - List<E> (implements)
  - RandomAccess (implements)
  - Cloneable (implements)
  - java.io.Serializable (implements)
- LinkedList is-a:
  - AbstractSequentialList<E> (extends)
  - List<E> (implements)
  - Deque<E> (implements)
  - Cloneable (implements)
  - java.io.Serializable (implements)

```java
package java.util;
public interface List<E> extends Collection<E> {
    // Query Operations
    int size();
    boolean isEmpty();
    boolean contains(Object o);
    Iterator<E> iterator();
    Object[] toArray();
    <T> T[] toArray(T[] a);

    // Modification Operations
    boolean add(E e);
    boolean remove(Object o);

    // Bulk Modification Operations
    boolean containsAll(Collection<?> c);
    boolean addAll(Collection<? extends E> c);
    boolean addAll(int index, Collection<? extends E> c);
    boolean removeAll(Collection<?> c);
    boolean retainAll(Collection<?> c);
    void clear();
    //...

    // Comparison and hashing
    boolean equals(Object o);
    int hashCode();

    // Positional Access Operations
    E get(int index);
    E set(int index, E element);
    void add(int index, E element);
    E remove(int index);

    // Search Operations
    int indexOf(Object o);
    int lastIndexOf(Object o);
```

Recap

- Interfaces

- Constants
  - Using
- Implementing
  - Closable
- Extending
  - Openable
- Multiple Interfaces
  - Openable, Lockable
- Code Reuse
  - Comparable
- List<E>
  - Is-a many things