# Interfaces

**Classes comprised exclusively of Abstract methods and constants**

**Polymorphism provides flexibility to interfaces**

## What is an Interface?

- It is a class containing methods without implementation!
  - Like an Abstract class on steroids!

```
//A class that a software
designer wants implemented
public interface Doable {
   public void doThis();
   public int doThat(int num);
   public boolean doTheOther();
}
```

Think of it as a **contract**
between
the designer of a class
and an implementor

An interface make it possible for a service to be available to a wide set of classes

## What is an Interface?

- A Java *interface* is composed of a collection of abstract methods and constants

**interface**
is a reserved word

Since all methods in an interface are abstract, the keyword `abstract` is left off

```
public interface Doable {
   public void doThis();
   public int doThat(int num);
   public boolean doTheOther ();
}
```

None of the methods in an interface are given a definition (body)

A semicolon immediately follows each method header

# Implementing Interfaces

```
// Clients can use this file
// without seeing the code
public interface Doable {
   public void doThis();
   public int doThat(int num);
   public boolean doTheOther();
}
```

```
// coders can implement this file
// without bothering Clients
public class CanDo implements Doable
{
   public void doThis ()
   {
      // code to do this
   }

   public void doThat (int num)
   {
      // code to do that
   }

   public boolean doTheOther ()
   {
   // whatever
   }
}
```

# Why we need Interfaces?

```java
// A class that a software designer wants implemented
// She wants clients to be able to use without seeing the code
// (they should just see only the INTERFACE to the class)
// She wants coders to update without messing up clients
// (coders should have freedom on how to IMPLEMENT it best)
public class Doable
{
   public void doThis ()
   {
      // code to do this
   }

   public void doThat (int num)
   {
      // code to do that
   }

   public boolean doTheOther ()
   {
      // whatever
   }
}
```

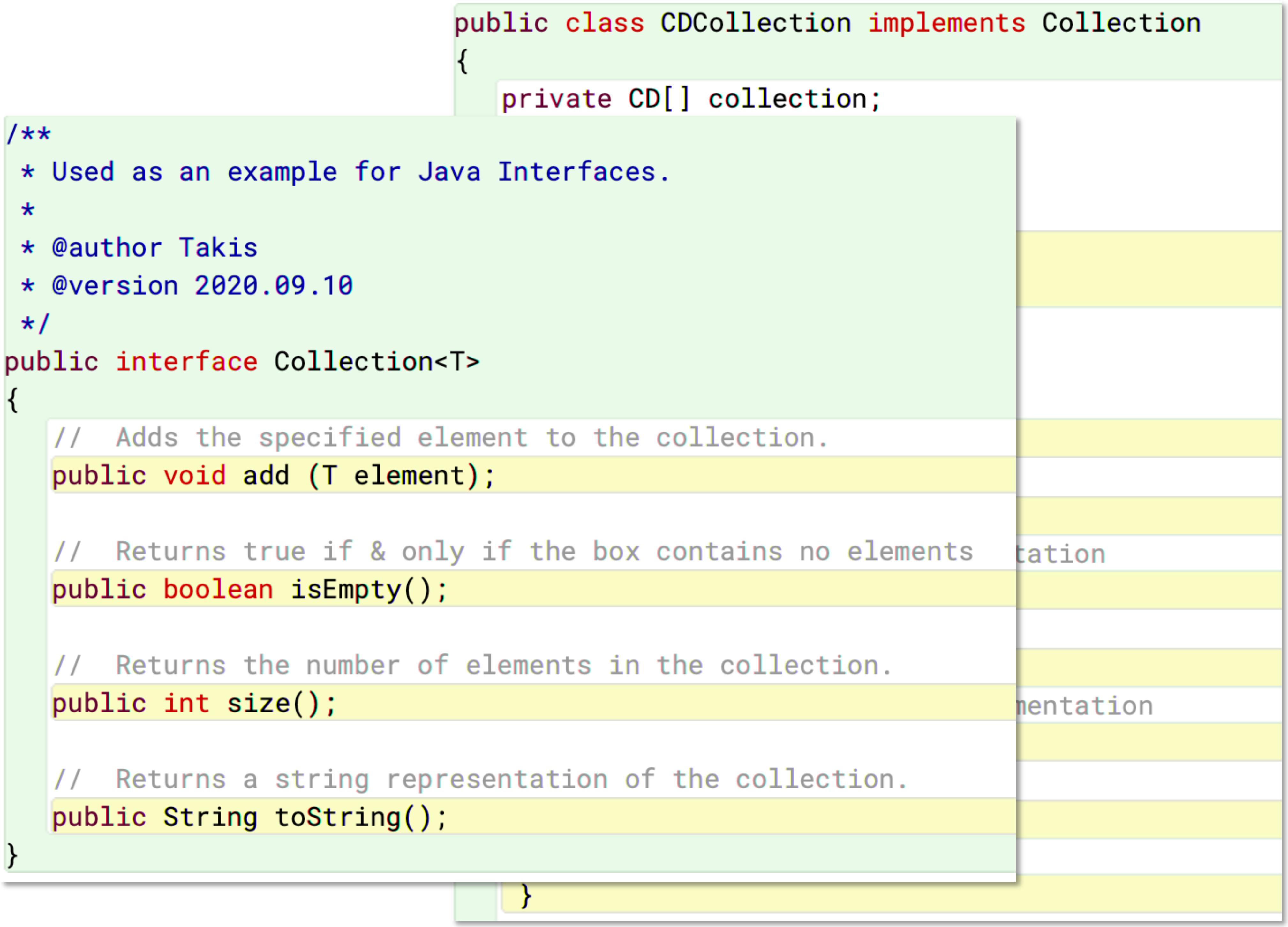
# Implementing Interfaces

`implements`
is also reserved word

```
public class CanDo implements Doable
{
   public void doThis ()
   {
      // code to do this
   }

   public void doThat (int num)
   {
      // code to do that
   }

   public boolean doTheOther ()
   {
   // whatever
   }
}
```

Each method listed in **Doable** is given a definition

- Why may an interface **not** be instantiated?
- Why are interface methods **public** by default?
- Why must a class implementing an interface, define **all methods** in the interface?
- Why may a class implementing an interface also implement other methods?



```java
public class CDCollection implements Collection
{
    private CD[] collection;
```

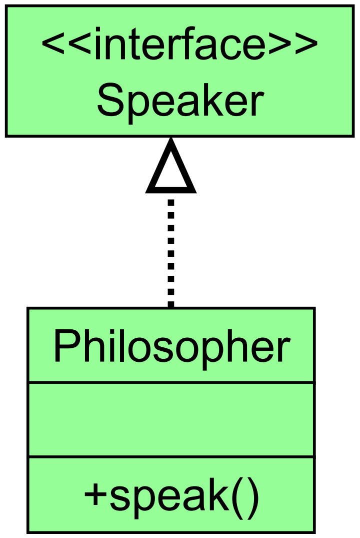
```
tation
```

```
mentation
```

```
    }
```

```java
/**
 * Used as an example for Java Interfaces.
 *
 * @author Takis
 * @version 2020.09.10
 */
public interface Collection<T>
{
    //  Adds the specified element to the collection.
    public void add (T element);

    //  Returns true if & only if the box contains no elements
    public boolean isEmpty();

    //  Returns the number of elements in the collection.
    public int size();

    //  Returns a string representation of the collection.
    public String toString();
}
```

# Polymorphism via Interfaces



- An interface name can be used
  as the type of an object reference variable

```
Speaker current;
```

- The current reference can be used
  to point to any object of any class
  that implements the Speaker interface

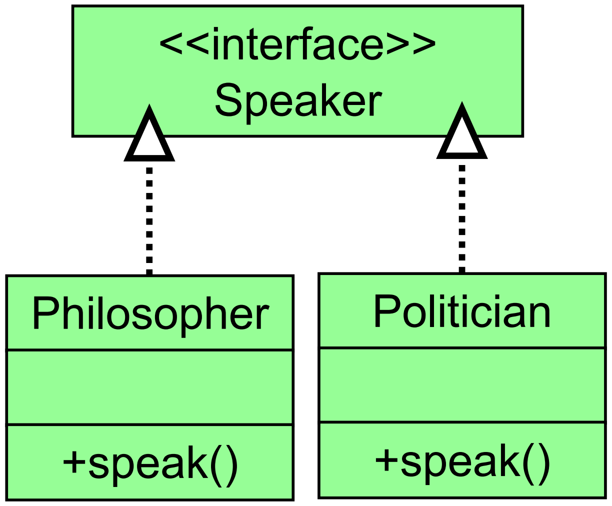
- The version of speak that the following line invokes
  depends on the type of object that current is referencing

```
current.speak();
```

# Polymorphism via Interfaces

- Suppose two classes, Philosopher and Politician, both implement the Speaker interface, providing distinct versions of the speak method
- In the following code, the first call to speak invokes one version and the second invokes another

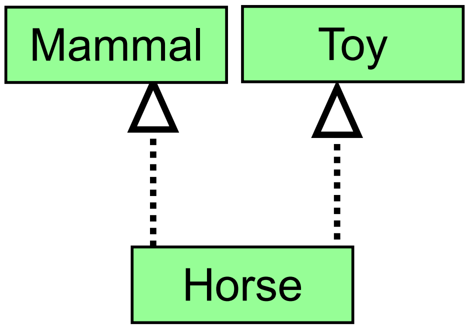
```
Speaker guest = new Philospher();
guest.speak();
guest = new Politician();
guest.speak();
```



## Multiple Interfaces

- A class can implement multiple interfaces
- The interfaces are listed in the `implements` clause
- The class **must** implement all methods in all interfaces listed in the header

```
class Horse implements Mammal, Toy {
   // all methods of both interfaces
}
```

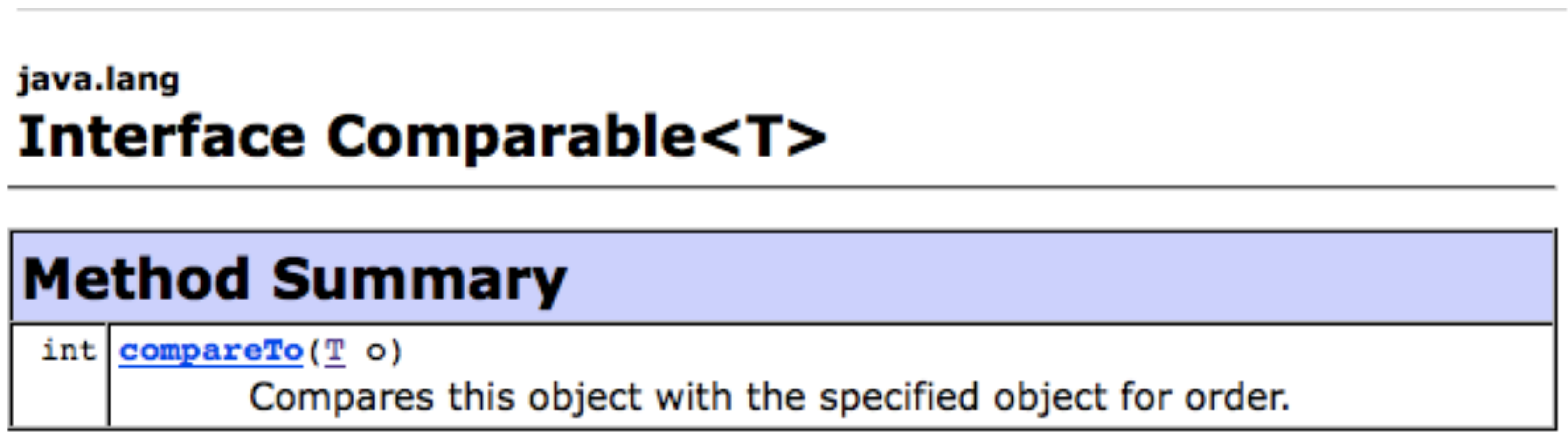


# The Comparable and The Iterable Interfaces

**Famous Java-defined interfaces**

# Java Interfaces: Comparable

- The Java standard class library has many helpful interfaces

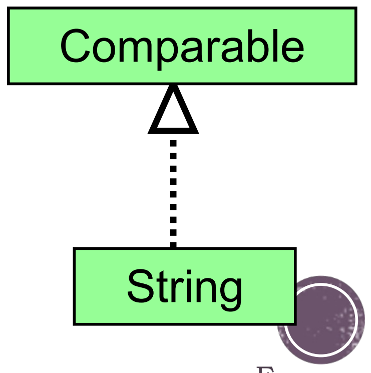


java.lang

**Interface Comparable<T>**

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

- The **Comparable** interface contains one abstract method called **compareTo**, which is used to compare two objects
- The **String** class implements **Comparable**, giving us the ability to put strings in lexicographic order



## The Comparable Interface

- Any class can implement Comparable to provide a mechanism for comparing objects of that type

```
 if (obj1.compareTo(obj2) < 0)
    System.out.println ("obj1 is less than obj2");
```

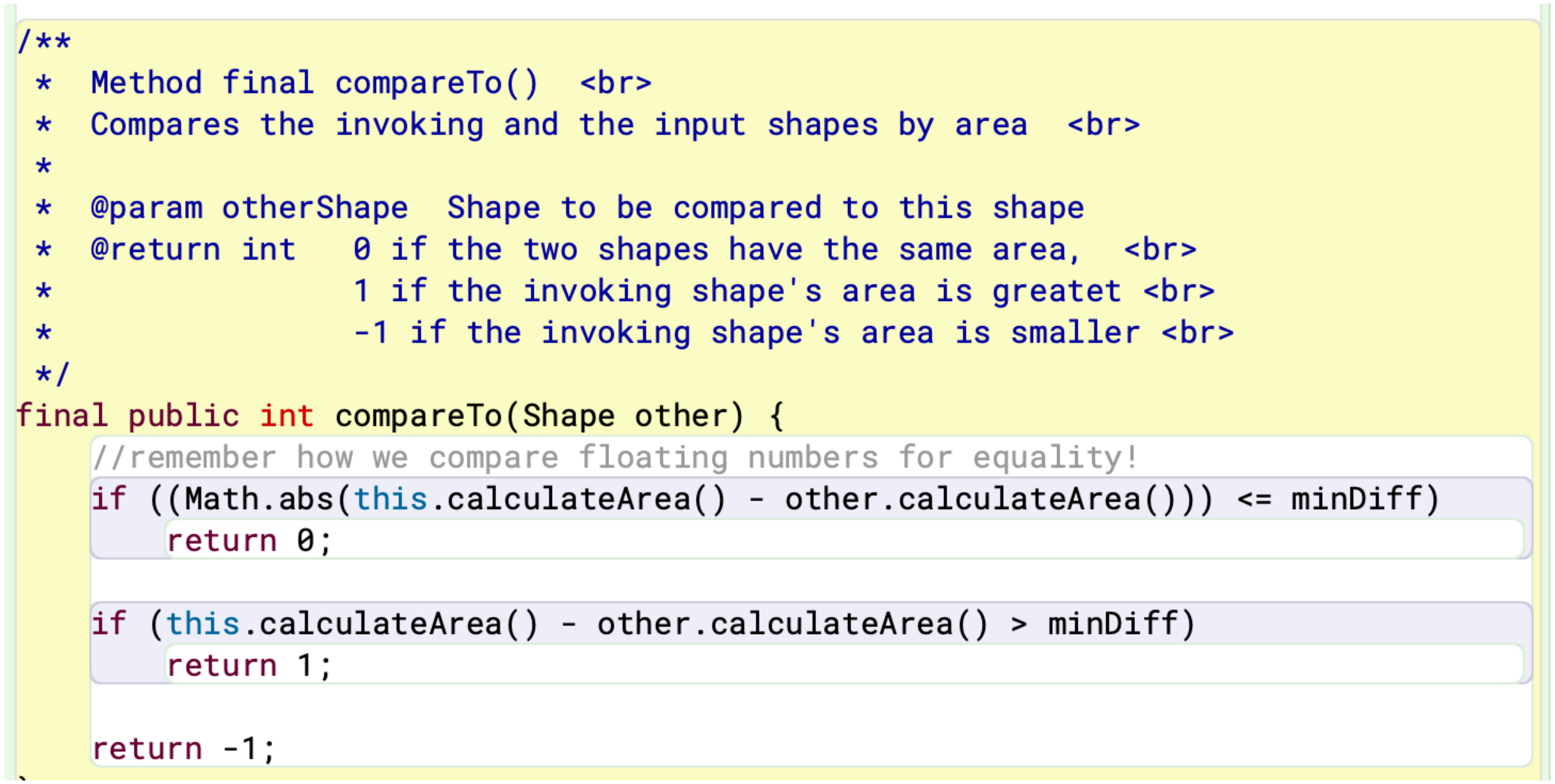
- It’s up to the programmer to determine what makes one object < than another
- You may define the compareTo method of an Employee class to order employees by name (alphabetically) or by employee number
- The implementation of the method can be as straightforward or as complex as needed for the situation

## String implements Comparable

- The String class contains a method called compareTo to determine if one string comes before another
- A call to `name1.compareTo(name2)`
  - returns 0 if name1 and name2 are equal (contain the same characters)
  - returns a negative value if name1 is less than name2
  - returns a positive value if name1 is greater than name2

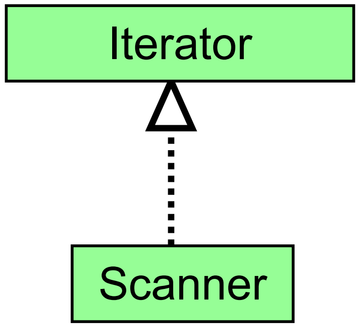
```
if (name1.compareTo(name2) < 0)
   System.out.println (name1 + "comes first");
else
   if (name1.compareTo(name2) == 0)
      System.out.println ("Same name");
   else
      System.out.println (name2 + "comes first");
```

# Shape implements `Comparable`

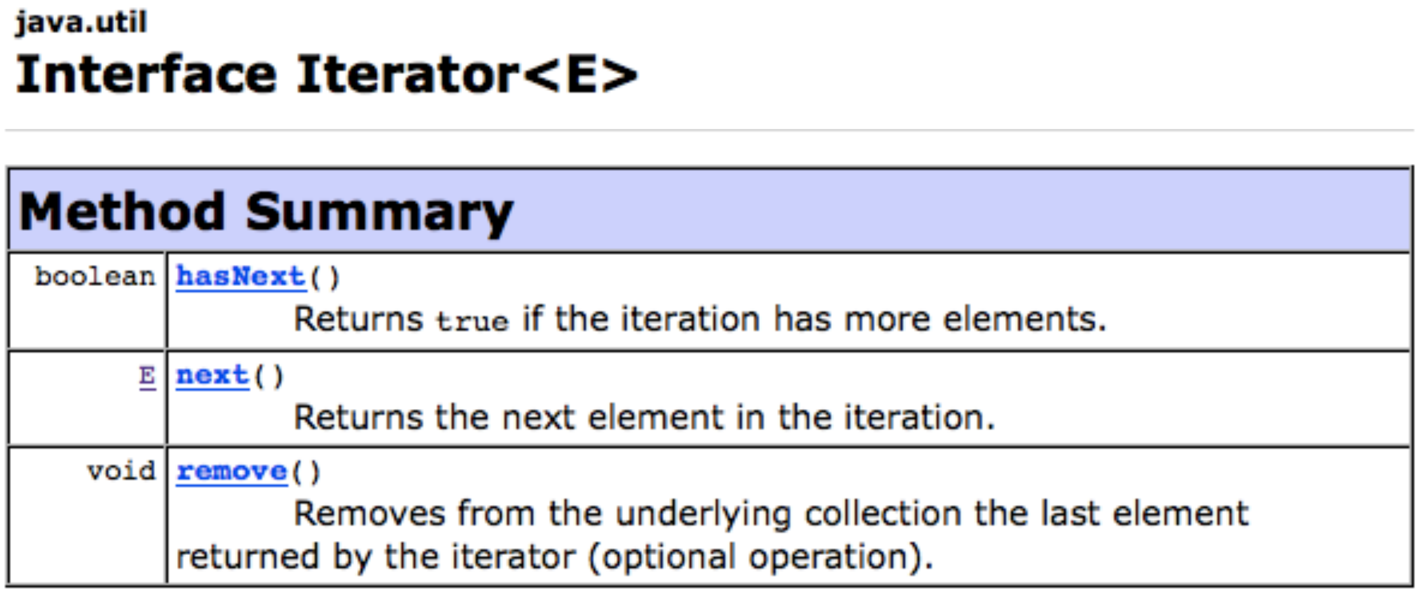


```java
/**
 *  Method final compareTo()  <br>
 *  Compares the invoking and the input shapes by area  <br>
 *
 *  @param otherShape  Shape to be compared to this shape
 *  @return int   0 if the two shapes have the same area,  <br>
 *                1 if the invoking shape's area is greatet <br>
 *                -1 if the invoking shape's area is smaller <br>
 */
final public int compareTo(Shape other) {
    //remember how we compare floating numbers for equality!
    if ((Math.abs(this.calculateArea() - other.calculateArea())) <= minDiff)
        return 0;

    if (this.calculateArea() - other.calculateArea() > minDiff)
        return 1;

    return -1;
```

# The Iterator Interface



- An iterator is an object that provides a means of processing a collection of objects, one at a time
- It is created formally by implementing the Iterator interface's 3 methods
  - The **hasNext** method returns a boolean – true if there are items left to process
  - The **next** method returns the next object in the iteration
  - The **remove** method removes the object most recently returned by the next method



java.util

**Interface Iterator<E>**

| Method Summary | |
|---|---|
| boolean | **hasNext**() Returns true if the iteration has more elements. |
| E | **next**() Returns the next element in the iteration. |
| void | **remove**() Removes from the underlying collection the last element returned by the iterator (optional operation). |

- By implementing the Iterator interface, a class formally establishes that objects of that type are iterators
- Once established, the for-each version of the for loop can be used to process the items in the iterator