ALGONQUIN COLLEGE

MODULE 09: INTERFACES AND COLLECTIONS

Professor : Dave Houtman
Office: T323
Office Hrs: Friday 11:30 – 12:45
Email: houtmad@algonquincollege.com

9.0 What is an interface?

Each reference type falls into one of three categories, shown below:

a) (arrays)
b) Classes
c) Interfaces

Simple *arrays* form the first type. They may be thought of as primitive versions of full-blown *classes*, which make up the second category of reference types.

In this module we explore the last of these three types, **interfaces**.

9.0 What is an interface?

Before addressing the question of what an interface *is*, recall the effect of the word `abstract` when used with classes and methods:

- *An abstract class is a class that cannot be instantiated*. You must inherit from it first using `extends`; the subclass can then be instantiated. An abstract class's *methods* may or may not be abstract, but a single abstract method will automatically make its class abstract.

  But the class can also be abstract *without* any of its methods being abstract. The concrete methods of an abstract class will be inherited by the subclass(es) and can be used the same as if inherited from a regular (concrete) superclass.

- *An abstract method must be overridden in its first concrete subclass*. In other words if the subclass is abstract, then you are *not* required to override abstract methods inherited from the superclass. But the first non-abstract class in a hierarchy of abstract subclasses must override the abstract methods it inherits.

9.0 What is an interface?

The two most important features of an interface are:

1. An interface is treated like a special kind of class in Java, one that consists *entirely of constants and abstract methods*. Therefore

   a) it cannot be instantiated. As with an abstract class, there's no such things as a `new` interface; and

   b) *all* of its methods must be overridden in the first concrete subclass

2. An interface acts like something of a contract; once you specify that a class **implements** an interface, you are guaranteeing that that class *must* contain concrete versions of every method specified in the interface. And then any instance of that class is guaranteed to have those features.

So the purpose of an interface is to declare a common set of behaviours for its derived classes, and therefore, any objects instantiated them. Any concrete class that implements an interface is guaranteed to fulfill a particular set of requirements, as specified by the interface's methods. And if *your* code implements an interface, then *you* must overload all the interface's methods with your own.

9.0 What is an interface?

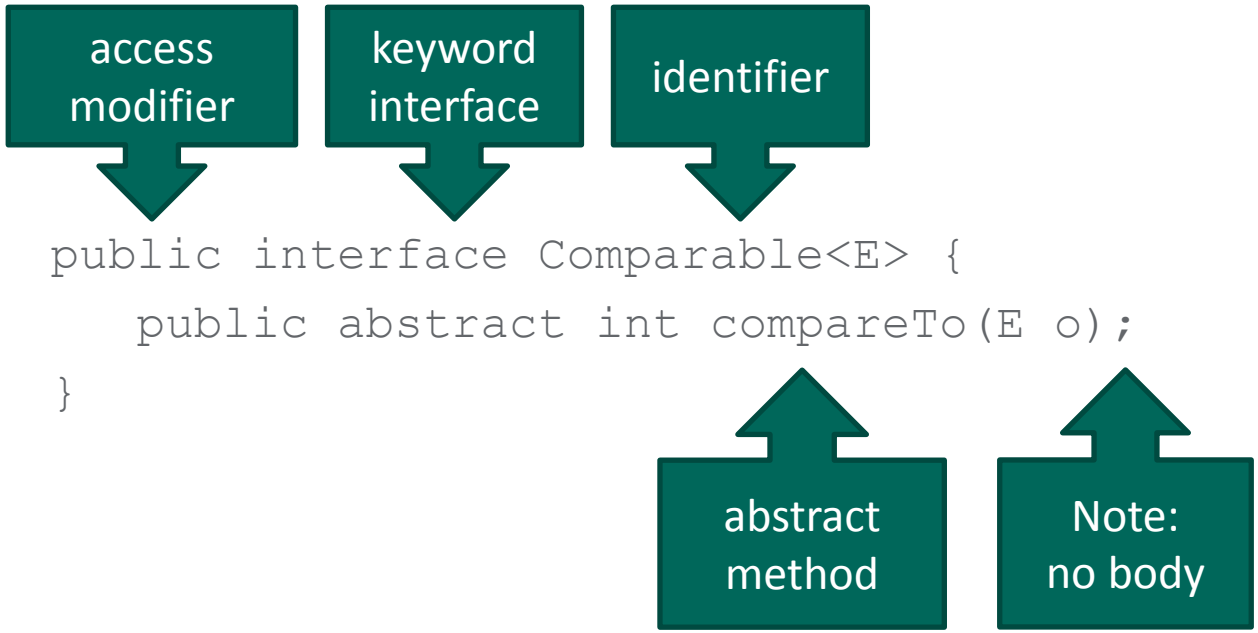
The form of an interface is similar to a class, except the word `interface` replaces the word `class`:

```
[access modifier] interface identifier{

   public static final fieldIdentifier1;    // constant properties
   public static final fieldIdentifier2;
   ...
   public abstract returnType methodIdentifier1();  //methods
   public abstract returnType methodIdentifier2();
   ...
}
```

9.0 What is an interface?

For example:



9.0 What is an interface?

When a class implements this interface, e.g.

```
public class MyNewClass implements Comparable<String>
```

then that class will need to implement the abstract method `compareTo()`, which is parameterized (in this case) with the `String` type. So `MyNewClass` will need to provide its own version of `compareTo()`:

```
public class MyNewClass implements Comparable<String>{
    ...
    @Override
    public int compareTo(String str){
       … // insert comparison code here
     }
}
```

9.0 What is an interface?

So implementing `Comparable<E>` guarantees that `MyNewClass` will have a `compareTo()` method available for use. The generic type `E` determines what kind of object the `compareTo()` method will be parameterized on.

Similarly, the declaration for the `EventHandler` interface is:

```
public interface EventHandler<T extends Event> {
   public abstract void handle (T event);
}
```

Any class that implements `EventHandler<T extends Event>` is guaranteed to provide a valid `handle()` method.

9.1 Abstract Classes v. Interfaces

In almost every way, you can treat an interface like a class with abstract methods:

- As with abstract classes, you cannot instantiate a new object from an interface using the `new` operator;
- You must import interfaces the same way as you do for classes using, e.g. `import java.classname.*`
- You can use an interface as a data type in the declaration of identifiers, arrays, and parameters passed to functions
- Since *every* method in an interface is abstract, *every* method must be overridden in the derived class…unless the derived class is abstract itself
- You can use an interface for casting
- Just as classes inherit via *class inheritance*, interfaces inherit via *interface inheritance*. However, since the two are so similar, we often refer to both as just *inheritance*, thus inferring that this may involve either classes *or* interfaces. Rather than sub- and superclasses, we have subinterfaces and superinterfaces

9.1 Abstract Classes v. Interfaces

- Because an interface contains nothing but constants and abstract methods, you do not need to explicitly use the words `public static final` and `public abstract` in the declaration of your fields and methods; this is automatically assumed inside an interface. Thus you will see

```
public interface Comparable<E> {
    public int compareTo(E o);
}
```

rather than

```
public interface Comparable<E> {
    public abstract int compareTo(E o);
}
```

...since `abstract` is assumed

9.1 Abstract Classes v. Interfaces

There are a few differences between classes and interfaces:

- Interfaces use the word `implements` rather than `extends`. For example, given `InterfaceA`, you can write:

```
public class MyNewClass implements InterfaceA {
   …
}
```

- You can both *extends* a class and *implements* an interface, e.g.

```
public class MyNewClass extends BaseClass
                                    implements InterfaceA {
   …
}
```

Thus `MyNewClass` inherits the methods of both `BaseClass` and `InterfaceA`

9.1 Abstract Classes v. Interfaces

Note that the declaration

```
public class MyNewClass implements InterfaceA {
   …
}
```

is equivalent to

```
public class MyNewClass extends Object
                                    implements InterfaceA{
   …
}
```

9.1 Abstract Classes v. Interfaces

- You can add multiple interfaces by separating each interface with a comma:

```
public class MyNewClass implements InterfaceA, InterfaceB {
    …
}
```

As noted earlier, while a derived class can only extend from *one* base class, it can have *multiple* interfaces. Thus the problem of multiple inheritance *does not apply to interfaces.*

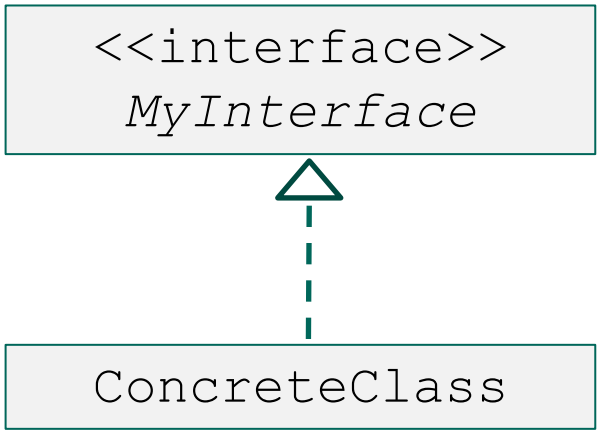
9.1 Abstract Classes v. Interfaces

- While interfaces are ‘implemented’ rather than ‘extended’, nonetheless the form of a bounded generic type for an interface uses ‘extends’, just as it would for a class. Thus

```
public class MyClass<T extends MyInterface>{
   //...
}
```

9.1 Abstract Classes v. Interfaces

- As with abstract classes, interface identifiers are italicized in UML. Unlike abstract classes, interface inheritance is symbolized using a dashed line



Note: the double angular brackets, `<< >>` , known as *guimettes*, are sometimes used to signal additional information in UML diagrams. This use is called a **stereotype**; the guimettes signal that we wish to extend UML beyond its regular use. In this case, the notation `<<interface>>` reminds us that we are not dealing with a class.

9.1 Abstract Classes v. Interfaces

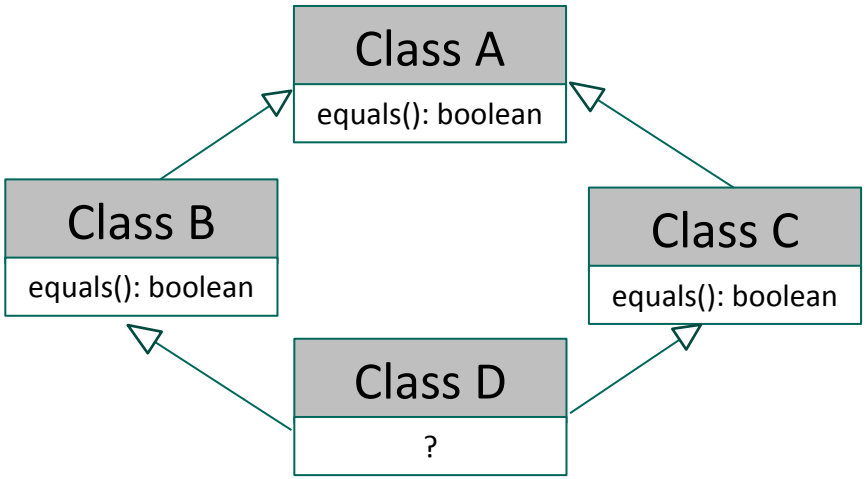
You can think of an interface as the purest expression of an abstract class—so pure, in fact, that it doesn't actually define any code itself, it only declares specifications for derived classes.

Supertype	Must implement/extend to a concrete class before object instantiation?	Must override *every* base class method in the derived class?
A concrete class	NO	NO
An abstract class with no abstract methods	YES	NO
A class with at least one abstract method	YES	NO only the abstract method(s) in the base class
An interface	YES	YES

9.1 Abstract Classes v. Interfaces

Given that interfaces and abstract classes are so much alike, a perfectly valid question to ask at this point is: what do you need interfaces for?

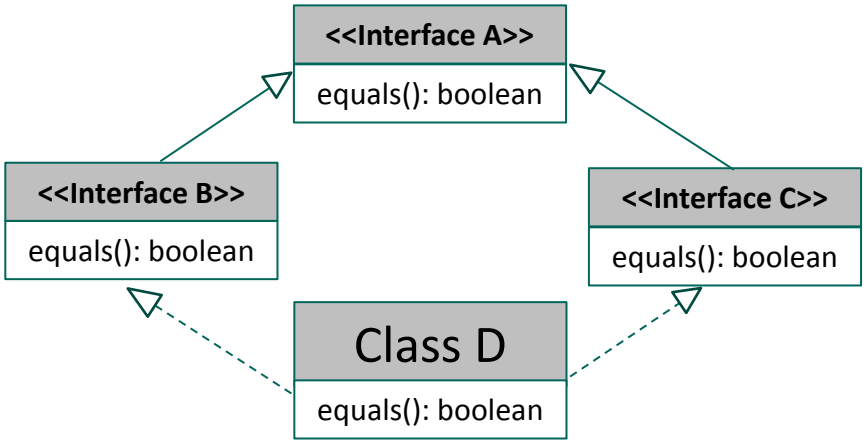
The answer takes us back to Module 04. Recall that Java does not permit **multiple inheritance**, since it leads to the **diamond problem**:



In the diagram above, Class D inherits from both B and C, and both of the latter have overridden A's `equals()` method. But whose `equals()` method does D use? The method could vary depending on the order in which the code from the two parent classes is compiled. This is obviously a dangerous way to compile code.

9.1 Abstract Classes v. Interfaces

Interfaces solve the diamond problem: this is their chief benefit over abstract classes (which, after all, are restricted by the rules of **single inheritance**). Consider Class D in the diagram below. It implements interface B and C, which both extend from A. But because neither A, B, nor C has a class body, D doesn't need to decide 'which' method to implement—there are no concrete methods to inherit from, and D is responsible for supplying its own implementation of `equals()`.



This may seem an awkward fix to the diamond problem, but it at least solves the problem while offering the programmer something like multiple inheritance.

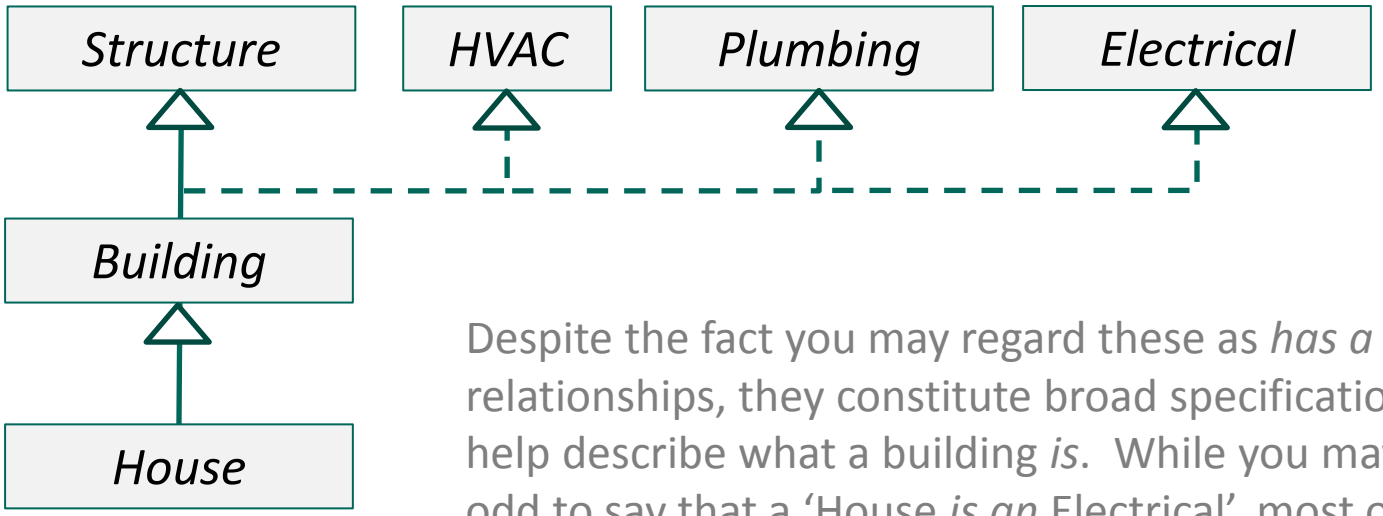
9.1 Abstract Classes v. Interfaces

Since an interface looks just like a purely abstract class, a reasonable question to ask is: when should I use one and when should I use the other?

In general, use inheritance from a class when there is a strong *is a* relationship: a dog *is a* animal, a house *is a* building. Interfaces should be used when there is a weak *is a* relationship, or even an abstract *has a* relationship, especially when a broad specification or behaviour is involved.

9.1 Abstract Classes v. Interfaces

For example, a building *is a* structure. However, any building must have certain core features—such as heating, plumbing and electrical facilities—which, while not a central to the definition of what a structure is, are essential to the definition of what a building *includes*. In the diagram below, the three interfaces straddle the line between an abstract *has a* feature and a weak *is a* (or *is part of*) feature.

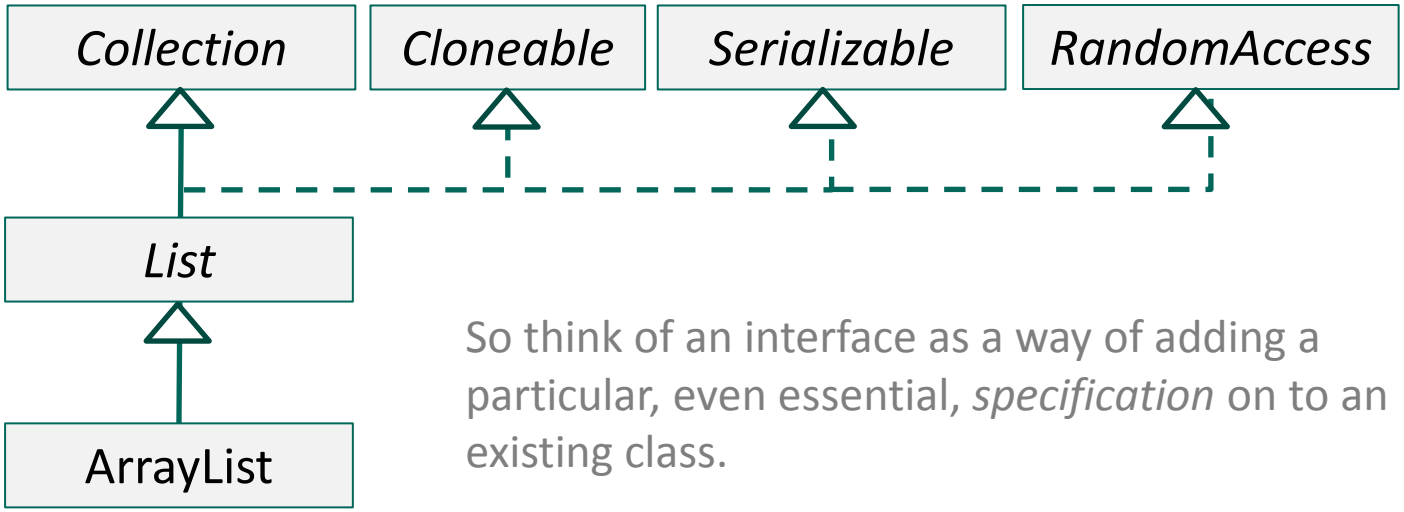


Despite the fact you may regard these as *has a* relationships, they constitute broad specifications that help describe what a building *is*. While you may find it odd to say that a 'House *is an* Electrical', most of us would agree that a modern house lacking electrical outlets lacks an essential feature of 'houseness'.

9.1 Abstract Classes v. Interfaces

In the world of programming, interfaces ensure that certain features are *guaranteed* to appear in subclasses (since their abstract methods *must* be overridden).

So think of interfaces as containing the specifications that must be tacked on to existing classes in order to guarantee their suitability for a particular task.



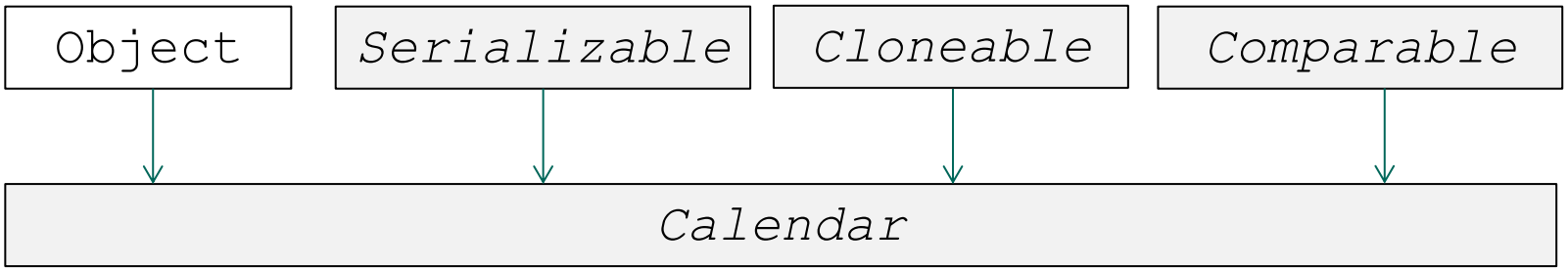
So think of an interface as a way of adding a particular, even essential, *specification* on to an existing class.

9.2 Implementing Interfaces

For example, in the case of the `Calendar` class

```
public abstract class Calendar
extends Object
implements Serializable, Cloneable, Comparable<Calendar>
```

this means that `Calendar` has added the methods in the `Serializable`, `Cloneable` and `Comparable` interfaces to the methods that would normally be available via `Object`:



9.2 Implementing Interfaces

Any class that extends `Calendar` will have the `Object` class's methods available to it (which may be overridden in `Calendar`), plus the methods found in the three interfaces, which are all overridden to concrete methods in `Calendar`.

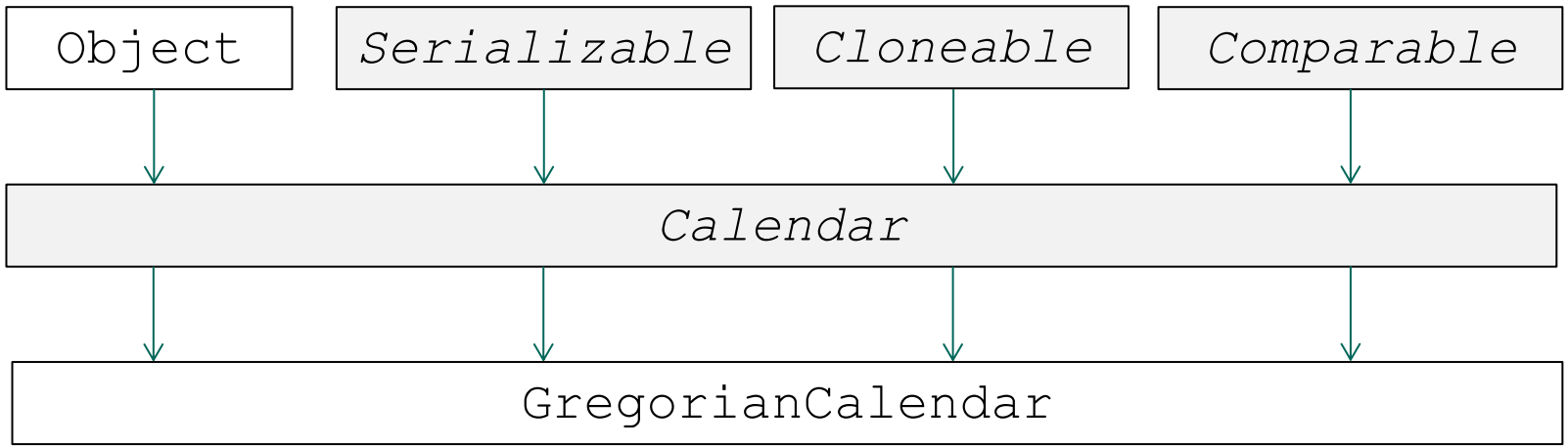
java.util

Class GregorianCalendar

java.lang.Object
    java.util.Calendar
        java.util.GregorianCalendar

All Implemented Interfaces:

Serializable, Cloneable, Comparable<Calendar>



9.2 Implementing Interfaces

Since the `Calendar` class implements `Comparable`…

```
public abstract class Calendar
extends Object
implements Serializable, Cloneable, Comparable<Calendar>
```

…and `Comparable` contains a single abstract method (seen earlier)…

```
public int compareTo(E o){
    … // insert comparison code here
}
```

…then `compareTo()` will be available to the user Calendar, and any of its subclasses, with `E` parameterized as a `Calendar` object.

int	compareTo(Calendar anotherCalendar) Compares the time values (millisecond offsets from the **Epoch**) represented by two `Calendar` objects.

9.2 Implementing Interfaces

Another example is `String`…

```
public final class String
extends Object
implements Serializable, Comparable<String>, CharSequence
```

…where again `Comparable` contains the same sole abstract method…

```
public int compareTo(E o){
    … // insert comparison code here
}
```

…which String *must* override. The method is then available to the user.

int	compareTo(String anotherString) Compares two strings lexicographically.

9.3 The Comparable and Comparator Interfaces

So any class that extends `Comparable` guarantees that it has a usable `compareTo()` method. The following are all classes that implement `Comparable`, and so each must implement its own `compareTo()` method internally:

```
public class Integer extends Number implements Comparable<Integer> {
    @Override
    public int compareTo(Integer o){
        // Uses the numerical value to return a int for sorting purposes
    }
}
```

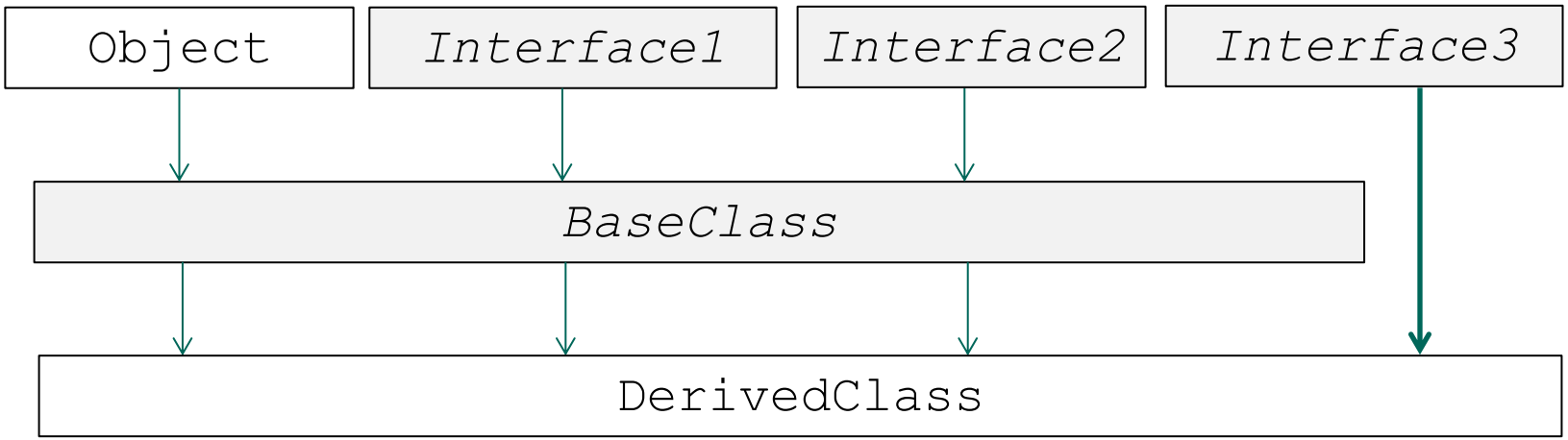
```
public class String extends Object implements Comparable<String> {
    @Override
    public int compareTo(String o){
        // Uses the character value to return a int for sorting purposes
    }
}
```

```
public class Date extends Object implements Comparable<Date> {
    @Override
    public int compareTo(Date o){
        // Uses the elapsed time to return a int for sorting purposes
    }
}
```

The above examples are derived from: Liang D. Y. (2014). *Introduction to Java Programming, Comprehensive Version, 10th Ed*. Toronto, ON: Pearson. pp. 509.

9.2 Implementing Interfaces

Recall that abstract methods do not need to be overridden in a subclass if that subclass is itself abstract. In such cases, any abstract methods not overridden in a superclass will need to be dealt with in the first concrete subclass.



This is exactly what happens with `EventHandler`. You implement it, and so you must override its `handle()` method in your (concrete) subclass.

9.3 The `Comparable` and `Comparator` Interfaces

Q. Why do I need `compareTo()`? What is it used for?

A. In order to sort objects into some kind of order, you need to be able to determine whether one object is greater than, less than, or equal to another. `compareTo()` performs this operation by comparing a particular property of two objects. The result of `compareTo()` *must* be an integer which is:

- A negative value if one object's property is less than another's
- A zero if the two object's properties are equal
- A positive value if one object's property is greater than another's

This numerical value is used by the sorting method to order the contents of the array/list.

Note that different objects will need to be sorted according to different features…

9.3 The Comparable and Comparator Interfaces

For example, when two `Calendar` objects are compared, the comparison is based on the time, in milliseconds, since Jan. 1, 1970 00:00:00.000 GMT. So the `compareTo()` method implemented by `Calendar` will look something like this:

```
@Override
public int compareTo(Calendar obj){
   return(this.getTimeInMillis() -
                              obj.getTimeInMillis());
}
```

This returns an integer value less than, greater than, or equal to 0, depending on whether `this` time is less than, greater than, or equal to the `obj`'s time, where `obj` is the other `Calendar` object being compared.

When you call `compareTo()` in an instantiated `GregorianCalendar` object, this is the operation you execute.

9.3 The Comparable and Comparator Interfaces

The integer returned by `compareTo()` is used to sort `Calendar` objects. For example, say we have an array of `GeorgianCalendar` objects:

```
Calendar[] arGC = new GregorianCalendar[10];
arGC[0] = new GregorianCalendar(); // Today's date
arGC[1] = new GregorianCalendar(1900, 1, 24);
arGC[2] = new GregorianCalendar(2042, 12, 31);
arGC[3] = new GregorianCalendar(1968, 11, 27);
// ...etc

Arrays.sort(arGC);  // uses compareTo() to sort dates
```

9.3 The Comparable and Comparator Interfaces

Similarly, `String` has its own version of `compareTo()`, which probably looks something like this:

```
@Override
public int compareTo(String obj){
   int i = 0;
   do {
      c1 = this.charAt(i);
      c2 = obj.charAt(i++);
   } while (c1 == c2);
   return (c1 - c2);
}
```

The loop exits the first time the two characters are not equal. Again, this returns an integer value less than, greater than, or equal to 0, depending on whether `this` String's current character is less than, greater than, or equal to the `obj`' String's equivalent character.

9.3 The Comparable and Comparator Interfaces

So we can load an array Strings and sort them as follows:

```
String[] cityNames = new String[]();
cityNames[0] = "Toronto, Canada";
cityNames[1] = "Sydney, Australia";
cityNames[2] = "Paris, France";
cityNames[3] = "Geneva, Switzerland";
// etc…

Arrays.sort(cityName);
```

The compareTo() method allows the sort method to operate by supplying an appropriate basis for comparisons between two String objects.

Note that since Strings and Calendar's are different objects, each needs to supply it's own compareTo() method, appropriate for the object itself: Strings are sorted by alphabetical order, Calendar objects are sorted by time.

9.3 The Comparable and Comparator Interfaces

What if you implement `Comparable` because *you* need to sort an array of objects? The you must override `compareTo()`. For example, assume you have created a `Complex` class designed to instantiate complex numbers. Assume that `Complex` has the method `modulus`, as follows:

```
public double modulus(){
   return Math.sqrt(getReal()*getReal() +
                            getImag()*getImag());
}
```

9.3 The Comparable and Comparator Interfaces

The modulus returns the magnitude of a complex number, and hence forms a natural basis for sorting such numbers. So after loading an array of `Complex` objects

```
Complex[] myComplexArray = {
    new Complex(5,1),
    new Complex("3+4i"),
    new Complex("2", "6")
};
```

we could use the `sort()` method on this array provided we had supplied a valid `compareTo()` method:

```
public class Complex implements Comparable<Complex>{
    @Override
    public int compareTo(Complex obj){
        return (this.modulus() - obj.modulus());
    }
    //etc.
}
```

9.3 The Comparable and Comparator Interfaces

In addition to Comparable, there is another interface that performs much the same function called (somewhat confusingly) Comparator. The definition for Comparator is:

```
public interface Comparator<T>{
   public int compare(T obj1, T obj2);
}
```

So while Comparable allows comparisons between this object with T, Comparator allows for comparisons within the same class of type T.

9.3 The `Comparable` and `Comparator` Interfaces

Q. Why have two ways to compare objects?

A. `Comparator` forces the user to use the *natural order* of the object: you must compare a particular feature of one object against a feature of an object *of the same class, T*.

With `Comparable`, you can compare features between *two* different classes. You could compare, say, the length of a `String` with the length of a `StringBuilder`. You can't do that with `compare()`, since only one object type, `T`, is available for comparison.

Whether a class implements one or the other interface is up to the designers of that class. Some classes implement *both*.

9.3 The `Comparable` and `Comparator` Interfaces

Notice that, regardless of which of the two methods you use, the mathematic result is always the same: return 0 if the two features of interest are equal, and a number greater or less than 0, if one feature is has greater or lesser priority than the other, in the sort order.

See: http://stackoverflow.com/questions/1440134/when-should-a-class-be-comparable-and-or-comparator, and http://www.javatpoint.com/difference-between-comparable-and-comparator

9.3 The Comparable and Comparator Interfaces

The `sort()` method is overloaded to allow us to use both `compare()` and `compareTo()`, depending on whether we wish to use the natural sorting order or not.

Note that `sort()` enforces its requirements by specifying that the argument of the method be a `Comparator` object (and hence one containing a valid `compareTo()` method)

static <T> void	sort(T[] a, Comparator<? super T> c) Sorts the specified array of objects according to the order induced by the specified comparator.

Similarly, for `ArrayList`, it's `sort()` method is declared as follows:

void	sort(Comparator<? super E> c) Sorts this list according to the order induced by the specified **Comparator**.

9.3 The `Comparable` and `Comparator` Interfaces

It's important at this point to recall what an interface is and why we use it:

The purpose of an interface is to declare
a common set of behaviours for its derived classes,
and therefore, any objects instantiated from them

Thus when we implement `Comparable` or `Comparator` with a class, we are signaling that it has appropriate features that allow the class to be sorted, i.e. it has a valid `compareTo()` or `compare()` method.

9.3 The Comparable and Comparator Interfaces

As an example, consider the following class:

```
public class CityInfo {
  private Integer population;
  private String name;
  private Float averageIncome;
  private Integer rating;

  public CityInfo(String name, int pop, float inc,int rat){
      setPopulation(pop); setName(name);
      setAverageIncome(inc); setRating(rat);
  }

  public void setPopulation(int pop){this.population= pop;}
  public int getPopulation(){return this.population;}

  public void setName(String name) {this.name = name;}
  //all other setters and getters, etc…
}
```

9.3 The Comparable and Comparator Interfaces

In a separate class, we add various `CityInfo` objects to an `ArrayList`:

```
ArrayList<CityInfo> cities = new ArrayList<>();

cities.add(new CityInfo("Vancouver, CA", 2313300, 67090.0f, 3));
cities.add(new CityInfo("Melbourne, AU", 4087100, 65500.0f, 1));
cities.add(new CityInfo("Vienna, AT", 1741200, 69200.0f, 2));
cities.add(new CityInfo("Toronto, CA", 2615000, 68110.0f, 4));
```

We may wish to sort the list by any of it's properties: name, average income, population, or rating. Thus the object passed to sort will differ depending on our sorting criteria.

9.3 The Comparable and Comparator Interfaces

We can accommodate these different sorting methods by constructing different classes, each of which implements `compare()` in different ways. For example

```
public class ByPopulation implements Comparator<CityInfo>{
   @Override
   public int compare(CityInfo c1, CityInfo c2) {
      return (c1.getPopulation() - c2.getPopulation());
   }
}

public class ByIncome implements Comparator<CityInfo>{
   @Override
   public int compare(CityInfo c1, CityInfo c2) {
      return (c1.getIncome() - c2.getIncome());
   }
}

// etc…
```

9.3 The Comparable and Comparator Interfaces

We can then choose to sort based on which comparator object we wish to use. To sort the cities array by population, we'd use

```
cities.sort(new ByPopulation());
```

and to sort by average income, use instead:

```
cities.sort(new ByIncome());
```

9.3 The Comparable and Comparator Interfaces

Note that our comparators can be implemented as inner classes. However, when using inner classes, we need to be careful to instantiate the outer class first, before using the inner class comparator in the `sort()` method.

```
public class CitySort{
   public static void main(String [] args){
      ArrayList<CityInfo> cities = new ArrayList<>();
      cities.add(new CityInfo("Vancouver, Canada", 2313300,
         67090.0f, 3));
      cities.add(new CityInfo("Melbourne, Australia", 4087100,
         65500.0f, 1));
      CitySort cs = new CitySort();  // instantiate this object
      cities.sort(cs.new ByPopulation());
   }
   public class ByRating implements Comparator<CityInfo> {
      @Override
      public int compare(CityInfo c1, CityInfo c2) {
         return (c1.getRating() - c2.getRating());
      }
   } ...
```

9.3 The Comparable and Comparator Interf... ⌘

void	**sort(Comparator<? super E> c)** Sorts this list according to the order induced by the specified **Comparator**.

Finally, note the generic notation used with `sort()`. The Comparator can take an element `E` or any of its *superclasses*. (This notation may be considered the compliment to `<E extends SuperClass>` which says: "E or any of its *sub*classes").

Why limit the comparison to superclasses, and not subclasses? If subclasses were allowed, the method used in the comparison could potentially be overridden in one of the subclasses, and the sort wouldn't work correctly. By specifying 'this class or higher', we ensure that all classes can only call the latest (i.e. overridden in superclasses, but not below the current class) methods.

9.4 Marker Interfaces: `Cloneable`

Two other interfaces are of particular interest to us: `Cloneable` and `Serializable`.

To be `Cloneable` means that a class can use `Object`'s `clone()` method. Not all classes can use `clone()` (perhaps it was overridden in the superclass) so `Cloneable` serves to indicate that `clone()` is still available.

As before, some methods demand that a class be `Cloneable` because they need to make a copy of the object. They will flag an error if the class is either not an `instanceof Cloneable`, or polymorphically incompatible.

9.4 Marker Interfaces: `Cloneable`

The `Cloneable` interface has the definition:

```
public interface Cloneable {}
```

In other words, it has *no* abstract methods to override. You may then well ask: of what use is `Cloneable`?

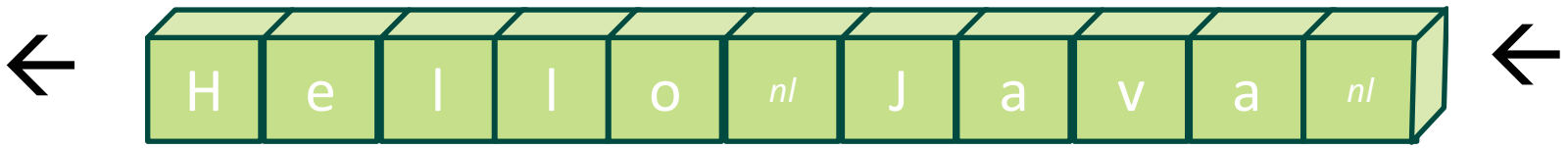
In this particular case, the fact that a class implements `Cloneable` is enough to indicate what the class is capable of. Interfaces used in this way, with no abstract methods, are called ***marker interfaces***. The `Cloneable` interface's association with any object indicates that the object can be cloned.

(The alternative would be to have a `Cloneable` interface with a method that returned `true` or `false` to indicate if a class could be cloned or not. But why bother, when the very presence of the interface itself is enough to tag an object as cloneable? That's what marker interfaces do.)

This also means that, effectively, any concrete class that implements a marker interface is *not* required to override its methods—there aren't any!

9.4 Marker Interfaces: `Serializable`

Recall from an earlier module that a *stream* is (essentially) a string of bytes 'flowing' through memory. Or more accurately, a *stream* is a data structure that keeps track of the start and end of a contiguous array of bytes in RAM. Special variables point to these locations and help keep track of which pieces of information are available to be read to or written to, whether they are 'flowing' in or out of the stream, whether the stream is buffered, etc.



9.4 Marker Interfaces: `Serializable`

Streams are an extremely important concept in data I/O. They are available for bytes, characters and strings. And objects.

Objects present something of a problem. Whereas characters are always two bytes long, and strings are terminated by CR, how does one determine where an object starts and ends in a stream? If you ship a Java applet over the internet to a web site, how does the browser know how to unpackage the bytes it receives from the input stream into objects?

9.4 Marker Interfaces: `Serializable`

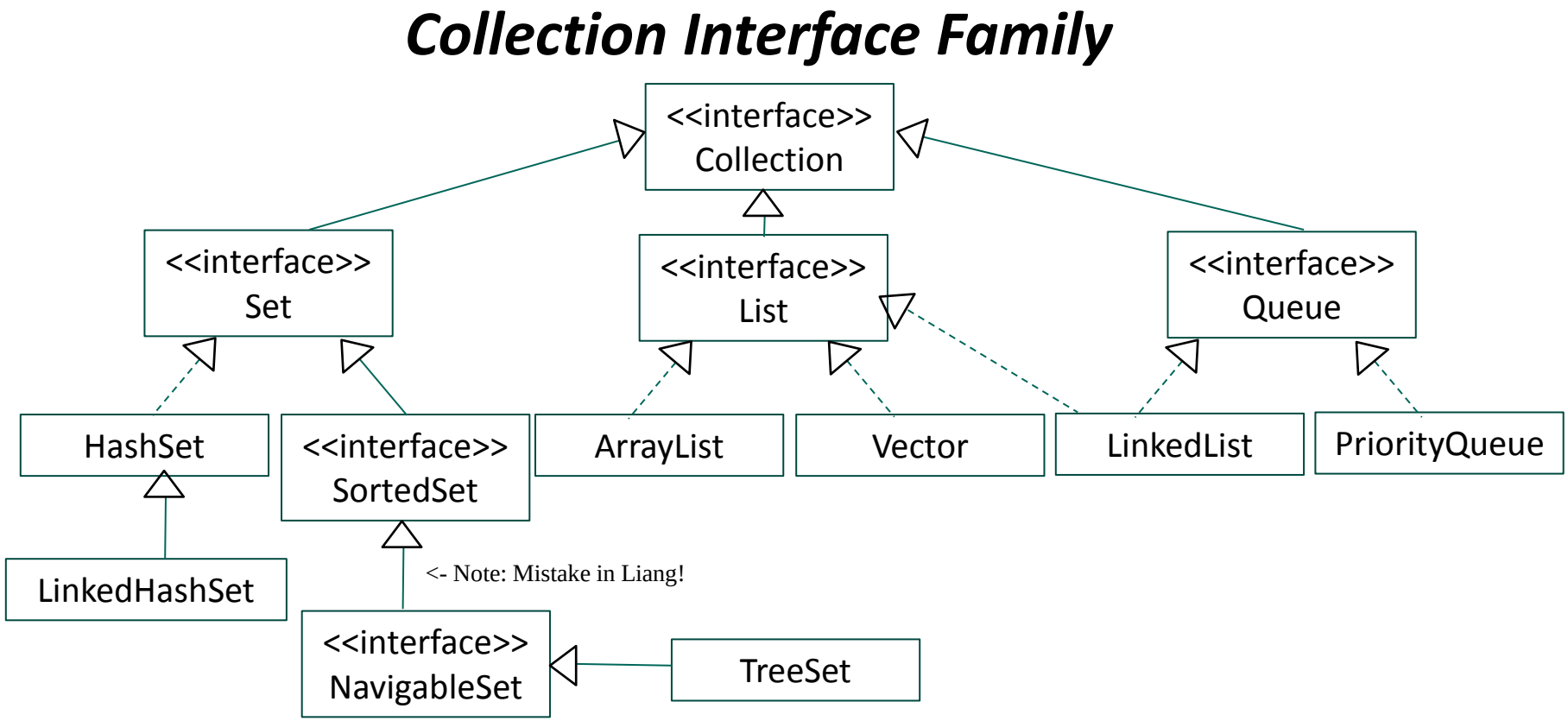
In Java, in order to create an object stream, it must first be made ***serializable***. Or in other words, packaged in such a way as to be shipped in a stream. For this purpose, there is a `Serializable` interface. Any time objects need to be transported, for example to and from a file, they need to implement `Serializable` first. This is a way of signalling that the information can be transported reliably.

Most objects in the Java API implement `Serializable`, including `String`, `StringBuilder`, `Date`, and `ArrayList`. But when you wish to store a custom-built object stream to or from a file, you must signal that it is `Serializable` first.

Like `Cloneable`, `Serializable` is a marker interface.

9.5 The `Collection` Interface

The `Collection` interface hierarchy forms a particularly rich family of interfaces. Here's a partial Java Collections Framework UML class diagram:



From: http://www.marcus-biel.com/wp-content/uploads/2015/08/class-and-interface-hierarchy.png

9.5 The `Collection` Interface

Given the following information from the Sun web page* *each* of the implementing classes (at the bottom) will provide access to the methods in the Collection interface, i.e. `add()`, `equals()`, `isEmpty()`…

Overview Package **Class** Use Tree Deprecated Index Help

Prev Class **Next Class** Frames No Frames All Classes

Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util

Interface Collection<E>

Type Parameters:

E - the type of elements in this collection

All Superinterfaces:

Iterable<E>

All Known Subinterfaces:

BeanContext, BeanContextServices, BlockingDeque<E>, BlockingQueue<E>, Deque<E>, List<E>, NavigableS

All Known Implementing Classes:

AbstractCollection, AbstractList, AbstractQueue, AbstractSequentialList, AbstractSet, ArrayBlockingQueue, Arra
ConcurrentLinkedQueue, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, DelayQueue,
LinkedList, LinkedTransferQueue, PriorityBlockingQueue, PriorityQueue, RoleList, RoleUnresolvedList, Stack,

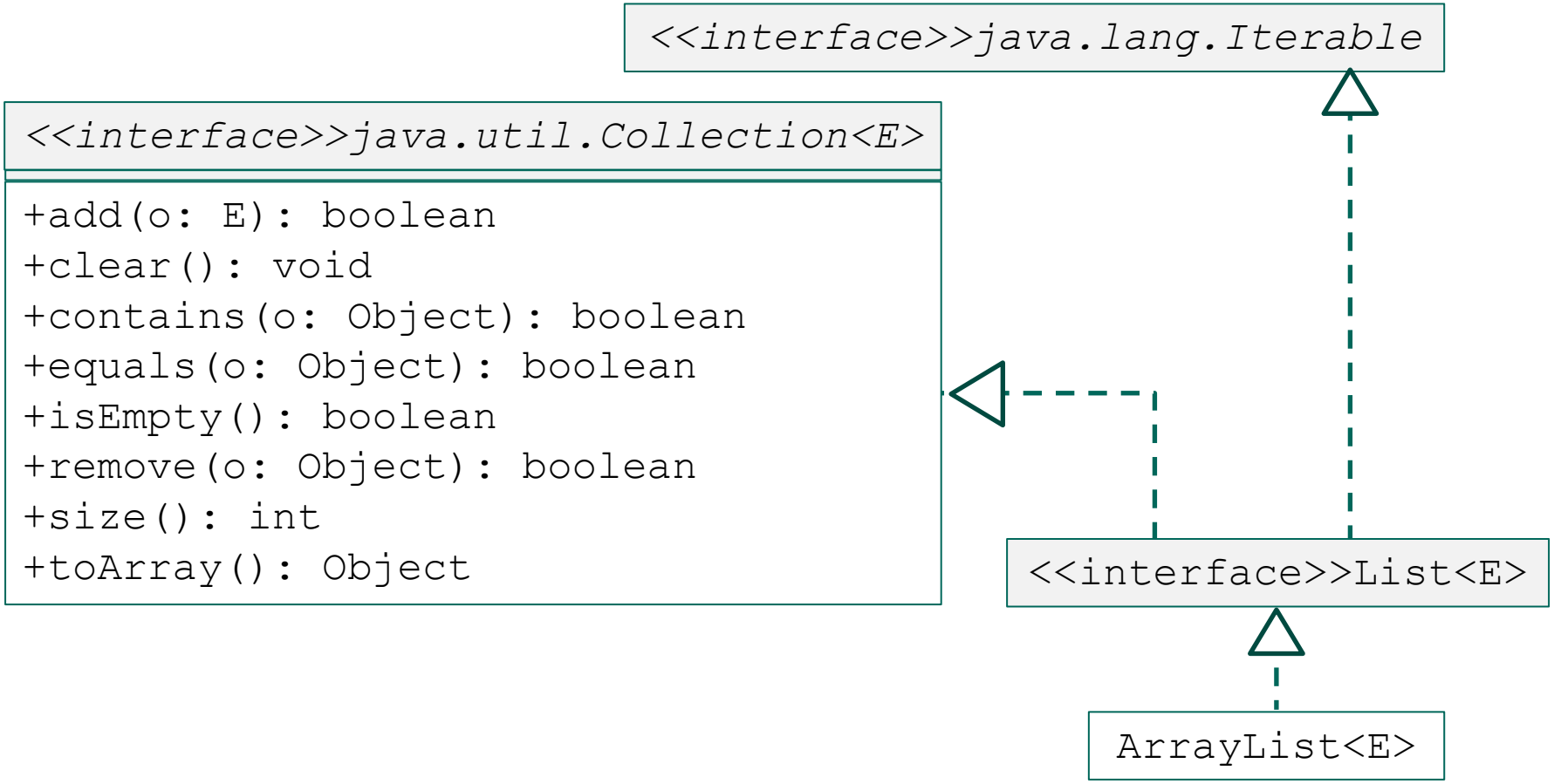
java.util.Collection<E>

```
+add(o: E): boolean
+clear(): void
+contains(o: Object): boolean
+equals(o: Object): boolean
+isEmpty(): boolean
+remove(o: Object): boolean
+size(): int
+toArray(): Object
```

*See: http://docs.oracle.com/javase/7/docs/api/java/util/Collection.html

9.5 The Collection Interface

So any subclass or subinterface, like `ArrayList`, will inherit all of the Collection interface's abstract methods. Being the first non-abstract class, `ArrayList` must provide concrete interpretations of all the abstract methods it inherits.



9.5 The `Collection` Interface

Hence `ArrayList()` must inherit, via the `List` interface, all of the methods found in the `Collection` interface.

Indeed, *any* class that implements the `Collection` interface is effectively saying:

"I make available the standard Collection methods"

Knowing that '*`ArrayList` implements `Collection`*' tells us `ArrayList` *makes the Collection interfaces methods available for use*.

Similarly, if you need to ensure that the `Collection` object's methods will be available for use, you should parameterize that method/class appropriately:

```
public static <T extends Collection> int myMethod(T obj1)…
```

9.5 The `Collection` Interface

So the various subinterfaces of `Collection` are each guaranteed to contain all of the `Collection` methods. But which of the *other* methods are passed down depends on which route we take through the inheritance hierarchy of the Collection Interface Family. And this depends on which features an object requires.

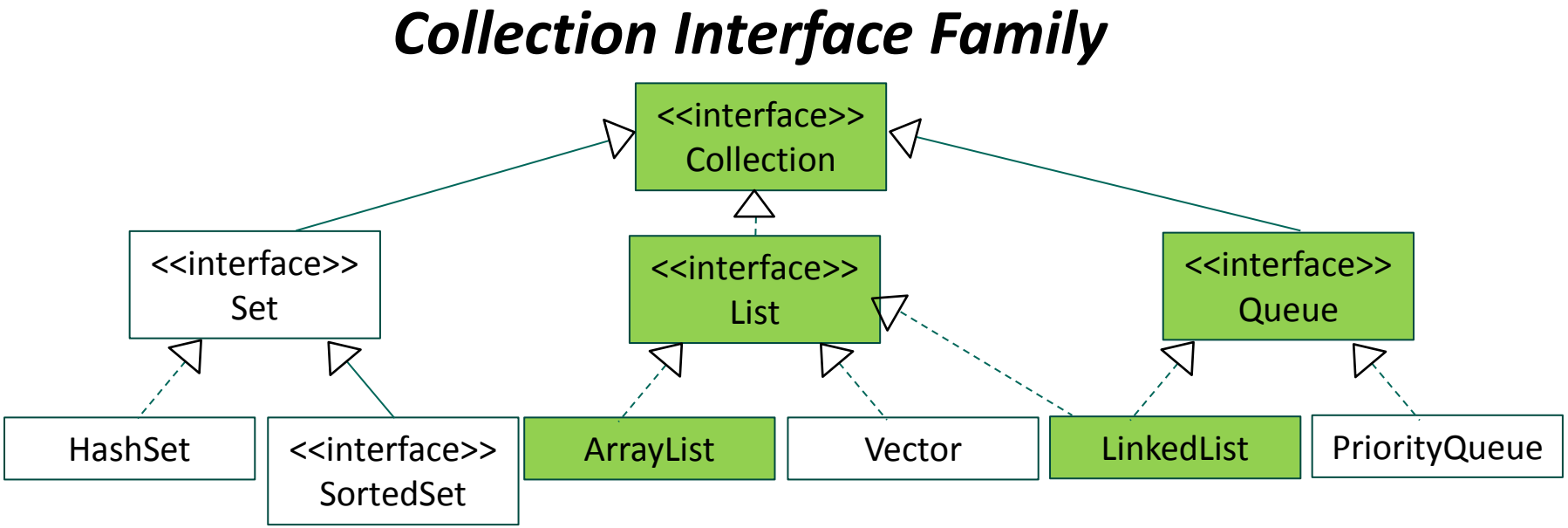
For example, an `ArrayList` has the following interfaces:

All Implemented Interfaces:

```
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess
```

9.5 The `Collection` Interface

Similarly a `LinkedList`—a type of data structure—implements both the `Queue` and `List` interfaces, which in turn implement the `Collection` interface.



9.5 The `Collection` Interface

So different interface specifications may be used to support alternate concrete implementations, i.e. different methods will be available in the *subinterfaces* according to the methods available in the *superinterfaces* . The presence of an interface in an object's family tree serves as a guarantee that specific public methods will be available to any calling method.

9.5 The Collection Interface

As a final comment on the Collection interface, note that it is not to be confused with the Collections class. The Collections class is an utility class (like Arrays) that consists of nothing but static methods. Amongst the most useful of its methods is the sort() method, which is overloaded.

```
Collections.sort(List<T extends Comparable> list)
```

takes any List object (including ArrayList, a subclass, as just shown) and sorts it according to T's compareTo() method. Additionally there is

```
Collections.sort(List<T> list, Comparator<? Super T> c)
```

Here, the second parameter is a Comparator that can be used to override the natural order of the sort given in the first parameter. So Collections.sort() provides the best features of both Arrays.sort() and the ArrayList.sort() in a single overloaded static method.

Questions

1. When talking about computer hardware, we frequently use the term 'interface' to refer to physical compatibility between connectors. How is this usage similar to the way the term is used in Java?
2. At the Oracle website, look up the interfaces used in the implementation of `ArrayList`. What specific purposes do you expect they serve?
3. `Collections.sort()` is declared to take a `List<T>` object as its argument, but can handle an `ArrayList<T>` as well. Which feature of OOP makes this possible?
4. Look at the diagram showing the Collection Interface family on slide 51. Notice that, according the this UML diagram, interfaces can both `extend` and `implement` other interfaces. What is the difference between
   `interfaceA` **`extends`** `interfaceB`, and
   `interfaceA` **`implements`** `interfaceB`?
5. Interfaces do not have a superparent the way classes do (with `Object`). However, every interface has access to `toString()`, `equals()`, etc. Why?