Interface

Contract between the class and the interface. The class agrees to add functionality to the methods included in the interface.

Interfaces

An interface is a class-like construct that contains only constants and a list of abstract methods

–These are methods that you may want to implement in a class

Syntax:

```
modifier interface InterfaceName {
  /** Constant declarations */
  /** Abstract method signatures */
}
```

Example:

```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

Interfaces

- Example: Suppose that you want to implement classes of Vehicles. The classes could be things like Car, Plane, Bicycle, Boat, etc. Whatever they are, they have to be able to do some things: stop(), go(), accelerate(), decelerate(), turn()

- You could decide to put all these methods into an interface

```
public interface VehicularActions {
public void go();
public void accelerate( double rate );
public void stop();
public void turn( boolean direction, int degrees);}
```

- Then if you chose to create a Boat class

```
public class Boat implements VehicularActions {...
```

Interface Members Modifiers

•Each field in an interface is implicitly ***public, static, and final***. Therefore, we don't need to use those modifiers.

•Methods in an interface are implicitly **public and abstract**. Therefore, we don't need to use those modifiers.

```
public interface T {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

```
public interface T {
  int K = 1;

  void p();
}
```

How do you know if a class implements an interface?

- A class implements an interface by
  - Declaring that it implements the interface
  - Defining implementations for all the interface methods

- A class may implement multiple interfaces by:
  - Listing the interfaces (comma separated) that it implements
  - Defining implementations for *all* of the interface methods

Example:

```
public class ClassName implements Interface1, Interface2 {...}
```

How do you know if a class implements an interface?

- If a class states in its definition line that it implements an interface, that means that it implements all of those methods.

  – If it doesn't, the compiler will detect it.

- You can find out whether or not a given object is of a class that implements a given interface by using the instanceof() method

  – InstanceOf() method: a boolean method that returns true if the class implements the interface, and false if it does not.

```
String s = "Hello.";
```

....< far, far away in another method>

```
if ( s instanceof String )
        System.out.println (s);
else
      //Do something else
```

Using a Generic Interface

- Insert the type inside the <angle brackets>

- Syntax:

- public class **ClassName** implements **InterfaceName** <**DataType**>

Example:

means its Sortable totally

public class Book implements Comparable<Book>

public class Phonebook implements Comparable<Phonebook>

Using an Interface as Type of a Parameter

- When an interface name is used as the type of a method's parameter, the actual argument must be a data type that implements the interface.

```
public static <T> int smaller(T[] data, Comparable<T> target)
{   // The return value is the number of objects in the data array that
    // are less than the non-null target b (using b.compareTo to compare
    // b to each object in the data array).
    int answer = 0;

    for (T next : data)
    {
        if (b.compareTo(next) > 0)
        {   // b is greater than the next element of data.
            answer++
        }
    }
    return answer;
}
```

Some Popular Java Interfaces

- The Comparable Interface
- The Clonable Interface
- The Iterator Interface

Comparable Interface

- Suppose that we want to place some objects into a total ordering. Example: comparing object to sort objects in array

- We need some way to compare objects.
-

- The generic Java way to do that is to implement the java Comparable interface, and to write a compareTo method in each class whose objects we might like to order.

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

Comparable interface

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

- int lesser = this.compareTo(obj2);
- CompareTo gives you three choices
  - This less than obj2 (return negative int) *usually* -1
  - This equals obj2 (return zero)
  - This greater than obj2 (return positive int) *usually* 1
  - *No guarantee that the non-zero returns are -1 or 1.*

- Many Java API classes implement comparable
  - Wrapper classes
  - Date
  - String

Comparable Example

- Example

```
public class CompareToExample{
    public static void main(String args[]){
        String s1="hello";
        String s2="hello";
        String s3="meklo";
        String s4="hemlo";
        String s5="flag";
        System.out.println(s1.compareTo(s2));//0 because both are equal
        System.out.println(s1.compareTo(s3));//-5 because "h" is 5 times lower than
        "m"
        System.out.println(s1.compareTo(s4));//-1 because "l" is 1 times lower than
        "m"
        System.out.println(s1.compareTo(s5));//2 because "h" is 2 times greater
        than "f"
    }
}
```

Clonable Interface

- The Cloneable interface specifies that an object can be cloned.

- Allows to create a copy of an object.

```
public interface Cloneable {
}
```

← This interface is empty (no abstract method or field)

- An interface with an empty body is referred to as a marker interface.

Clonable Interface

- An interface with an empty body is referred to as a marker interface.
  - A marker interface does not contain constants or methods.
  - It is used to denote that a class possesses certain desirable properties.
  - A class that implements the Cloneable interface is marked **cloneable**, and
  - its objects can be cloned using the **clone()** method defined in the **Object** class.

Iterator interface

- An **iterator** is a way to step through all the elements of a data structure without having to expose the details of how data is stored in the data structure.

- Why use iterators?

– They make it easier to remove elements while iterating through a list.

–When you're removing elements from a collection, you have to be very careful not to try to read beyond the ever-changing last index of the collection. Iterators handle this (capacity-checking operations) problem for you.

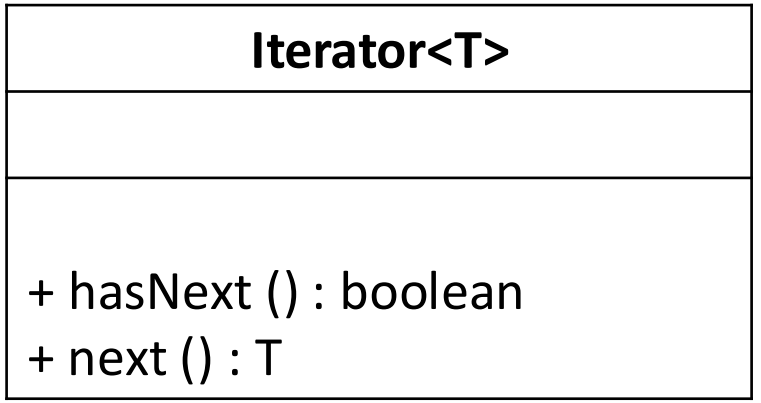
–Using an Iterator can also be sometimes faster than "growing your own" loop to iterate over a collection.

Iterator interface

- The Iterator interface provides a uniform way for traversing elements in various types of collections.

- Iterators are usually implemented using generics.

```
public interface Iterator<E> {
public Boolean hasNext();
public E next();
public void remove(); // optional
```



- The remove method doesn't always make sense.

–For instance, on a read-only collection.

Iterator interface

- Every class in the Java Collections Framework implements Iterator.
  - They provide an iterator() method that returns an instance of an Iterator over the elements in that collection.

- Iterator vs Iterable Interface
  - There is also a separate Iterable interface that allows an object to be the target of an enhanced for statement (the “for-each” loop).
  - This is something different, we don’t talk about it here.

Subclasses of Iterator

- You can see that there are subclasses of Iterator.

- ListIterator allows you to iterate through a List forwards or backwards.

- There are also Iterators for primitives.

- Lots of stuff you can do if you want to look up the API and play with it.
  - All are OK to use in this course.
  - None are required.

How to Traverse a List

- List is an interface that is implemented by things like
  - ArrayList<E>
  - LinkedList<E>

ArrayList <E>

- https://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

- What is it a list of?

- Objects vs primitive data types
- Autoboxing/Wrapper classes

Integer	Short
Double	Float
Boolean	Long
Character	Byte

Activity

- Look up the documentation of the *List* interface and provide a brief description of its methods. Then compare the implementations of those methods in *ArrayList* vs *LinkedList*