Module: Core Java

Session 20: Generics and Legacy Classes

Generics and Legacy classes

- Historical Collection Classes
    - Arrays
    - Vector and Stack Classes
    - Enumeration Interface
    - Dictionary, Hashtable, Properties Classes
    - BitSet Class
- Collections Framework Enhancements in J2SE 5
- Understanding Generics
    - Using the Enhanced For Loop
    - Primitive Data Types and Autoboxing
- Benefits of the Java Collections Framework

Objective:

At the end of this chapter, you will be able to:

- Implement following Historical Collection Classes
    - Arrays
    - Vector and Stack Classes
    - Enumeration Interface
    - Dictionary, Hashtable, Properties Classes
    - BitSet Class
- Implement Generics collections
- Understand importance of collection framework

Historical Collection Classes

There can be situations when you need to use some of the original collections capabilities rather than the new ones. The capabilities of working with some of these collections (arrays, vectors, hashtables, enumerations, and other historical capabilities.) is shown below:

Arrays

You must have learnt about arrays while learning the basics of Java programming language. According to their definition, Arrays are fixed-size collections of the same datatype. You must remember that only Arrays can store primitive datatypes. All others including arrays, can store objects. While creating an array, the number and type of object you wish to store has to be specified. The size of an array can neither grow nor store a different type. (unless it extends the first type).

To find out the size of an array, you ask its single public instance variable, length, as in array.length.

To access a specific element, either for setting or getting, you place the integer argument within square brackets ([int]), either before or after the array reference variable. The integer index is zero-based, and accessing beyond either end of the array will throw an ArrayIndexOutOfBoundsException at runtime. However, if a long variable is used to access an array index, you get a compiler-time error.

Arrays are full-fledged subclasses of java.lang.Object. They can be used with the various Java programming language constructs excepting an object:

Consider the following example to implement Array class:

```
        import java.util.*;
```

```
class ArrayDemo
{
public static void main ( String args[])
{
int array[] = new int [10];
for ( int i =1; i<10;i++)
array[i]= 3 * i ;

System.out.println(" Origina;l Content ") ;
display(array);
Arrays.sort(array);
System.out.println("Sorted : ") ;
display(array);
Arrays.fill(array,2,6,-1);
System.out.println("After fill() :") ;
display(array);
System.out.println("The value -9 is at location") ;
int index = Arrays.binarySearch(array,-9);
System.out.println(index) ;
}
static void display( int array[])
{
for ( int i=0;i<10;i++)
System.out.println(array[i]+" ") ;
System.out.println("") ;
}
}
```

When an array is created, it is automatically initialized, either to false for a Boolean array, null for an Object array, or the numerical equivalent of 0 for everything else.

To make a copy of an array, perhaps to make it larger, you use the arraycopy() method of System. In the destination array, you need to preallocate the space.

- `System.arraycopy(Object sourceArray, int sourceStartPosition, Object destinationArray, int destinationStartPosition, int length)`

Vector and Stack Classes

A Vector is a historical collection class whose size can grow, but it can store heterogeneous data elements. With the Java 2 SDK, version 2, List interface can be implemented by the Vector class as it has been retrofitted into the Collections Framework hierarchy. You should use ArrayList, if you are using the new framework.

When transitioning from Vector to ArrayList, one key difference is the arguments have been reversed to positionally change an element's value:

- From original Vector class
  `void setElementAt(Object element, int index)`
- From List interface
  `void set(int index, Object element)`

The Stack class extends Vector to implement a standard last-in-first-out (LIFO) stack, with push() and pop() methods. Be careful though. Since the Stack class extends the Vector class, you can still access or modify a Stack with the inherited Vector methods.

Let us see an example to understand the usage of **Vector class:**

```
//Vector Example
import java.util.*;
class vectordemo
{
public static void main(String args[])
{
Vector v = new Vector(3,2);
System.out.println("Initial size "+v.size());
System.out.println("Initial capacity "+v.capacity());
  v.addElement(new Integer(1));
  v.addElement(new Integer(2));
  v.addElement(new Integer(3));
  v.addElement(new Integer(4));
System.out.println("Capacity after addition "+v.capacity());
  v.addElement(new Double(50.20));
  v.addElement(new Double(22.56));
System.out.println("Capacity after addition "+v.capacity());
  v.addElement(new Float(0.20));
  v.addElement(new Float(2.56));
System.out.println("Capacity after addition "+v.capacity());
  v.insertElementAt( new Double(10.29),2);
  v.removeElementAt(2);
System.out.println("First Element "+(Integer)v.firstElement());
System.out.println("Last Element "+(Float)v.lastElement());
}
}
```

Let us see an example for the usage of **Stack Class**:

```
import  java.util.* ;
class StackDemo
{
 static void showpush( Stack st,int a)
{
st.push(new Integer(a));
System.out.println("push ( " +  a + " ) " );
System.out.println("Stack : " + st);
}
static void showpop(Stack st)
{
System.out.println("POP ---- > ");
Integer a = (Integer) st.pop();
System.out.println(a);
System.out.println("Stack : " +st );
}
public static void main(String args[])
{
Stack st = new Stack();
System.out.println("Stack  " + st );
showpush(st, 45);
showpush(st, 50 );
showpush(st, 65);
showpush(st,70 );
showpush(st,80);
showpop(st);
showpop(st);
showpop(st);
showpop(st);
showpop(st);
try
{
showpop(st);
}
catch (EmptyStackException e )
{
System.out.println("Empty Stack ");
}
}
}
```

Enumeration Interface

To iterate through all the elements of a collection, you can use Enumeration interface. This interface has been superceded by the Iterator interface in the Collections Framework. You may use Enumeration from time to time because all libraries do not support the newer interface.

- boolean hasMoreElements()
- Object nextElement()

Iterating through an Enumeration is similar to iterating through an Iterator, though method names are more preferred with Iterator. However, there is no removal support with Enumeration.

```
Enumeration enum = ...;
while (enum.hasNextElement()) {
  Object element = iterator.nextElement();
  // process element
}
```

Dictionary, Hashtable, Properties Classes

The Dictionary class is full of abstract methods. Infact, it should have been an interface. It forms the basis for key-value pair collections in the historical collection classes, with its replacement being Map, in the new framework. Hashtable and Properties are the two specific implementations of Dictionary available.

Storage of any object (except null) as its key or value is permitted with the Hashtable implementation and hence is called a generic dictionary. With the new version of Java SDK, to implement the Map interface, the class has been reworked into the Collections Framework. So, you can go for the original Hashtable methods or the newer Map methods. If you need a synchronized Map, using Hashtable is slightly faster than using a synchronized HashMap.

To work with text strings, a specialized Hashtable called Properties implementation is used. Properties class allows gets you text values without any casting whereas you have to cast values retrieved from a Hashtable to your desired class. Loading and saving property settings from and input stream or to an output stream is also supported by the class. System.getProperties() retrieves the system properties list which is the most commonly used set of properties.

Here’s an example to understand the usage of HashTable Class:

```
import java.util.*;
class hashtabledemo
{
public static void main(String args[])
{
//Create a hash table
Hashtable balance=new Hashtable();
Enumeration names;
String str;
//Put elements into ‘balance’
balance.put("Benjamin gilani",new Double(3434.34));
balance.put("Nusrulla Khan",new Double(7834.33));
balance.put("Benjamin gilani",new Double(3439.80));
balance.put("Arijit bolbaka",new Double(1378.00));
balance.put("Anu Kalia",new Double(299.34));
balance.put("Ali Kuli Khan",new Double(399.34));
balance.put("Golmal Khan",new Double(-19.34));
//show all balances in hashtable
names=balance.keys();
while(names.hasMoreElements())
{
str=(String)names.nextElement();
System.out.println(str+":"+balance.get(str));
}
System.out.println();

//Deposit 1000 into Benjamin gilani's account
double balance0=((Double)balance.get("Benjamin gilani")).doubleValue();
balance.put("Benjamin gilani", new Double(balance0+1000));
System.out.println("Benjamin gilani's new balance "+balance.get("Benjamin
gilani"));

//Deposit 2000 into Ali Kuli Khan's account
double balance1=((Double)balance.get("Ali Kuli Khan")).doubleValue();
balance.put("Ali Kuli Khan",new Double(balance1+2000));
System.out.println("Ali Kuli Khan's new balance "+balance.get("Ali Kuli
Khan"));

//Withdraw 1345 from Nusrulla Khan's account
double balance2=((Double)balance.get("Nusrulla Khan")).doubleValue();
balance.put("Nusrulla Khan",new Double(balance2-1345));
System.out.println("Nusrulla Khan's new balance "+balance.get("Nusrulla
Khan"));
}
}
```

BitSet Class

BitSet: It gives an alternate representation of a set. If a finite number of n objects are given, each object can be associated with a unique integer. Then each possible subset of the objects corresponds to an n-bit vector, with each bit "on" or "off" depending on whether the object is in the subset. For small values of n, a bit vector might be an extremely compact representation. However, for large values of n an actual bit vector might be inefficient, when most of the bits are off.

Collections Framework Enhancements in J2SE 5

1. Collections framework has been enhanced with the help of three new language features:
   - ➢ **Generics**
     - i. Adds compile-time type safety to the collections framework
     - ii. Eliminates the need to cast when reading elements from collections.
   - ➢ **Enhanced for loop**
     - i. When iterating over collections, it eliminates the need for explicit iterators
   - ➢ **Autoboxing/unboxing**
     - i. Automatically converts primitives (such as int) to wrapper classes (such as Integer) when inserting them into collections
     - ii. Converts wrapper class instances to primitives when reading from collections.

Understanding Generics

The most important feature added to Collections API is Generics. It plays and important role in enhanced for loop and autoboxing. Generics are Java's answer to C++ templates, but in many ways they are much more. A specific type can be associated with collections with the help of Generics, which was not possible before J2SE 5.0, and if tried to associate, it created many problems.

```
import java.util.*;
public class BasicCollection {
public static void main(String args[]) {
ArrayList list = new ArrayList();
    list.add( new String("One") );
    list.add( new String("Two") );
    list.add( new String("Three") );
    Iterator itr = list.iterator();
    while( itr.hasNext() ) {
      String str = (String)itr.next();
      System.out.println( str );
    }
  }
}
```

The preceding code shows how a typical collection class might look prior to J2SE 5.0. The class creates an ArrayList and adds three strings to it. Note that there's no specific type associated with this ArrayList; you can add any class that inherits from Object to the ArrayList.

Look at the *while* loop that retrieves the strings from the ArrayList.

```
while(itr.hasNext() ) {
   String str = (String)itr.next();
   System.out.println( str );
}
```

The loop iterates over every item in the ArrayList. But notice what you have to do for each element retrieved—*typecast* each element. The typecast is required because the ArrayList does not know what types it stores. This is the problem that generics solve in Java. Generics allow you to associate a specific type with the ArrayList. So when you upgrade this BasicCollection class to use generics, the problem disappears. The GenericCollection class shown below constitutes an upgraded version

```
import java.util.*;
  public class GenericCollection {
 public static void main(String args[]) {
   ArrayList<String> list = new ArrayList<String>();
   list.add( new String("One") );
   list.add( new String("Two") );
   list.add( new String("Three") );
   //list.add( new StringBuffer() );
   Iterator<String> itr = list.iterator();
   while( itr.hasNext() ) {
     String str = itr.next();
     System.out.println( str );
   }
 }
}
```

As you can see, only a few lines change when you upgrade the class to J2SE 5.0. The first is the line of code that declares the ArrayList; this version declares the ArrayList using a type of String.

```
ArrayList<String> list = new ArrayList<String>();
```

Notice how the code combines the type with the name of the collection. This is exactly how you specify all generics, delimiting the name of the type with angle brackets (<>), placed immediately after the end of the collection name.

Next, when declaring the Iterator, you declare it as an Iterator for the String type. You specify the generic type for the iterator exactly as you did for the ArrayList, for example:

```
Iterator<String> itr = list.iterator();
```

That line specifies the Iterator's type as String. Now, when you call the *next* method for the iterator, you no longer need the typecast. You can simply call the *next* method and get back a String type.

```
String str = itr.next();
```

Although that's a nice code-saving feature, generics does more than just save you from having to do a typecast. It will also cause the compiler to generate a compile error when you try to add an "unsupported" type to the ArrayList. What is an unsupported type? Because the ArrayList was declared to accept strings, any class that is not a String or a String subclass is an unsupported type.

The class StringBuffer is good example of an unsupported type. Because this ArrayList accepts only Strings, it will not accept a StringBuffer object. For example, it would be invalid to add the following line to the program.

```
list.add( new StringBuffer() );
```

Here's the real beauty of generics. If someone attempts to add an unsupported type to the ArrayList you won't get a runtime error; the error will be detected at compile time. You will get the following compile error.

```
c:\collections\GenericCollection.java:12:
   cannot find symbol
symbol  : method add(java.lang.StringBuffer)
location: class java.util.ArrayList<java.lang.String>
    list.add( new StringBuffer() );
```

Catching such errors at compile time is a huge advantage for developers trying to write bug-free code. Prior to J2SE 5.0 this error would have likely shown up later as a ClassCastException at runtime. Catching errors at compile time is always preferable to waiting for the right conditions at runtime to cause the bug, because those "right conditions" may very well appear only after deployment, when your users are running the program.

Using the Enhanced For Loop

Java did not support for each loop for some time. Using languages like Visual Basic and C#, looping through the contents of a collection was very easy because they supported a for loop. With the introduction of for each loop in collections, one can access the contents of a collection without the need for an iterator. Now (finally) Java has a *for each* looping construct, in the form of the enhanced *for* loop.

The enhanced *for* loop was one of the most anticipated features of J2SE 5.0. It simplifies your code to a greater extent since there is no need of an iterator. Here's a version, upgraded to use the enhanced *for* loop.

```
import java.util.*;
public class EnhancedForCollection {
  public static void main(String args[]) {
```

```
            ArrayList<String> list = new ArrayList<String>();
            list.add( new String("One") );
            list.add( new String("Two") );
            list.add( new String("Three") );
            //list.add( new StringBuffer() );
            for( String str : list  ) {
              System.out.println( str );
            }
          }
        }
```

The preceding code eliminates the Iterator and while loop completely, leaving just these three lines of code.

```
            for( String str : list  ) {
              System.out.println( str );
            }
```

The format for the enhanced for loop is as follows.

```
        for( [collection item type] [item access variable] :
           [collection to be iterated] )
```

The first parameter, the collection item type, must be a Java type that corresponds to the generic type specified when the collection was created. Because this collection is type ArrayList, the collection item type is String.

The next parameter provided to the enhanced for loop is the name of the access variable, which holds the value of each collection item as the loop iterates through the collection.

That variable must also be the same as that specified for the collection item type.
For the last parameter, specify the collection to be iterated over. This variable must be declared as a collection type, and it must have a generic type that matches the collection item type. If these two conditions are not met, a compile error will result.

Primitive Data Types and Autoboxing

Another feature added to J2SE 5.0 is the ability to automatically box and unbox primitive datatypes. Using this feature, the complexity is reduced to a great extent when you have to add and access primitive data types in the collections API.

To understand what boxing and unboxing are, have a look to how primitive data types in Java were handled prior to J2SE 5.0. Here's a simple Java application that makes use of primitive data types with the collections API, prior to J2SE 5.0.

```
     import java.util.*;


        public class PrimitiveCollection {
```

```
      public static void main(String args[]) {
        ArrayList list = new ArrayList();

        // box up each integer as they are added
        list.add( new Integer(1) );
        list.add( new Integer(2) );
        list.add( new Integer(3) );
        //list.add( new StringBuffer() );

        // now iterate over the collection and unbox
        Iterator itr = list.iterator();
        while( itr.hasNext() ) {
          Integer iObj = (Integer)itr.next();
          int iPrimitive = iObj.intValue();
          System.out.println( iPrimitive );
       }
      }
    }
```

The preceding code illustrates two distinct steps that had to occur when using primitives with the collections API. Because the collection types hold Objects rather than primitive types, you had to box primitive item types into a suitable wrapper object. For example, the preceding code boxes the int primitive data type into an Integer object using the following lines of code:

```
 list.add( new Integer(1) );
list.add( new Integer(2) );
list.add( new Integer(3) );
```

But the boxing requirement presents a problem when later code tries to access the primitive int variables stored in the collection. Because the primitive data types were stored as Integer objects they will be returned from the collection as Integer objects, not as int variables, forcing yet another conversion to reverse the boxing process:

```
 Iterator itr = list.iterator();
while( itr.hasNext() ) {
Integer iObj = (Integer)itr.next();
  int iPrimitive = iObj.intValue();
  System.out.println( iPrimitive );
}
```

In this case, the code first retrieves each item as an Integer object, named iObj. Next, it converts the Integer object into an int primitive by calling its intValue method.
Using autoboxing, storing primitive types in collections and retrieving them becomes far simpler. Here's an example.

```
 import java.util.*;
```

```
public class AutoBoxCollection {
  public static void main(String args[]) {
    ArrayList<Integer> list = new
      ArrayList<Integer>();

    // box up each integer as it's added
    list.add( 1 );
    list.add( 2 );
    list.add( 3 );
    //list.add( new StringBuffer() );

    // now iterate over the collection
    for( int iPrimitive: list  ) {
      System.out.println( iPrimitive );
   }
  }
}
```

This autoboxing example begins by creating an ArrayList with the generic type of Integer.

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

After creating the list with the wrapper type of Integer, you can add primitive ints to the list directly.

```
list.add( 1 );
list.add( 2 );
list.add( 3 );
```

Now, you no longer need to wrap each integer in an Integer object. In addition, accessing the individual list members is also considerably easier.

```
for( int iPrimitive: list  ) {
  System.out.println( iPrimitive );
}
```

The preceding example shows how you can use the enhanced for loop to iterate over a collection of primitive data types: in this case, retrieving each primitive variable directly as an int with no type conversion required. In other words, autoboxing and unboxing let you use primitive data types with collections as easily as objects.

With the addition of generics to J2SE 5.0, it is easy to specify what type of data a collection can store exactly. Thus the collection accepts only the correct type of object, and eliminates the need to typecast items stored to or retrieved from the collection. It is only the compiler that ensures these entire things.

In many programming languages you have found ‘for’ in each construct. Now the same functionality has been used in Generics that allow you to use the enhanced for loop. So by using “enhanced for loop" the need for iterators and the whole iterating process through a collection of items becomes simple and easy.

Finally, it is not an easy task to add primitive data types to collections. Developers have to handle the feature called Autoboxing and Unboxing where they have to box primitive types such as int into Integer objects and then unbox back into int primitive types. This results in much clearer source code while using collections with primitive data types.

However due to these changes, whatever technique you use to access collection data in J2SE 5.0 is thus been changed. But it has simplified Java code used to access the collections API to a great extent.

2. There are three new collection interfaces that are provided:
    - ➢ **Queue** – This represents a collection that is designed to hold elements before it goes for processing. Other than basic Collection operations, queues also provide additional insertion, extraction, and inspection operations.
    - ➢ **lockingQueue** – This is an extension of the Queue that waits for the queue to become non-empty while retrieving an element and also the wait for space to become available in the queue when an element is being stored. (This interface is included in package java.util.concurrent.)
    - ➢ **ConcurrentMap** – This extends Map with atomic putIfAbsent, remove, and replace methods. (This interface is included in package java.util.concurrent.)
3. There are two new concrete Queue implementations that are provided, where one existing List implementation has been retrofitted to implement Queue, and one abstract Queue implementation is provided:
    - ➢ **PriorityQueue** – It is an unbounded priority queue backed by a heap.
    - ➢ **ConcurrentLinkedQueue** – It is an unbounded thread-safe FIFO (first-in first-out) queue backed by linked nodes. (This class is part of java.util.concurrent.)
    - ➢ **LinkedList** – This is retrofitted to implement the Queue interface. LinkedList behaves as a FIFO queue, when accessed via the Queue interface.
    - ➢ **AbstractQueue** – An implementation of skeletal Queue.
4. There are five new implementations of BlockingQueue implementations that are provided. (All of these are included in java.util.concurrent):
    - ➢ **LinkedBlockingQueue** – It is a FIFO blocking queue, optionally bounded and backed by linked nodes.
    - ➢ **ArrayBlockingQueue** – It is a FIFO blocking queue, bounded and backed by an array.
    - ➢ **PriorityBlockingQueue** – It is an unbounded blocking priority queue backed by a heap.
    - ➢ **DelayQueue** – It is a time-based scheduling queue backed by a heap.
    - ➢ **SynchronousQueue** – It is a simple rendezvous mechanism utilizing the BlockingQueue interface.
5. There is one ConcurrentMap implementation:
    - ➢ **ConcurrentHashMap** – It is a ConcurrentMap implementation backed by a hash table highly concurrent and with high-performance. When this implementation is used, it never blocks at the time of retrieval and the client

can select the concurrency level for updates. This is done so as to have a drop-in replacement for Hashtable, which is in addition to implementing ConcurrentMap. It supports all of the "legacy" methods peculiar to Hashtable.

6. There are some special-purpose List and Set implementations provided for situations where read operations are more than write operations and also iteration cannot or should not be synchronized:
    - ➢ **CopyOnWriteArrayList** – It is a List implementation backed by an array. When you make a new copy of the array, all mutative operations (such as add, set, and remove) are implemented. No synchronization is required, even during iteration. Iterators promises never to throw ConcurrentModificationException. This implementation is best for maintaining event-handler lists (where it has infrequent change, and traversal is frequent and potentially time-consuming).
    - ➢ **CopyOnWriteArraySet** – It is a Set implementation backed by a copy-on-write array. This implementation is same as CopyOnWriteArrayList. The add, remove, and contains methods require time proportional to the size of the set which is unlike most Set implementations. This implementation is best for maintaining event-handler lists that must prevent duplicates.
7. There are special-purpose Set and Map implementations that are provided for use with enums:
    - ➢ **EnumSet** – It is a Set implementation backed by a bit-vector with a high-performance. All elements must be elements of a single enum type of each EnumSet instance.
    - ➢ **EnumMap** – It is a Map implementation backed by an array with a high-performance. All keys must be elements of a single enum type in each EnumMap instance.
8. For the use with generic collections a new family of wrapper implementations is provided:
    - ➢ **Collections.checkedInterface** – It returns a dynamically typesafe view of the specified collection, which throws a ClassCastException. It throws the same if a client wants to add an element of the wrong type. You are provided with compile-time (static) type checking in this generics mechanism but it is possible to defeat this mechanism in the language. This possibility is eliminated entirely by dynamically typesafe views.
9. There are three new generic algorithms and one comparator converter added to the Collections utility class:
    - ➢ **frequency(Collection<?> c, Object o)** - This counts the number of times the specified element occurs in the specified collection.
    - ➢ **disjoint(Collection<?> c1, Collection<?> c2)** – This determines whether they contain no elements in common or whether two collections are disjoint.
    - ➢ **addAll(Collection<? super T> c, T... a)** – This is convenient method where you can add all of the elements in the specified array to the specified collection.
    - ➢ **Comparator<T> reverseOrder(Comparator<T> cmp)** – A comparator is returned that gives you the reverse ordering of the specified comparator.

10. We have content-based hashCode and toString methods outfitted in the Arrays utility class for arrays of all types. The existing equals() methods is complemented by these methods. The versions of all the three methods are provided that operate on nested (or "multidimensional") arrays. They are deepEquals, deepHashCode, and deepToString. It is not very necessary to print the contents of any array.

    The idiom for printing a "flat" array is:

    ```
    System.out.println(Arrays.toString(a));
    ```

    The idiom for printing a nested (multidimensional) array is:

    ```
    System.out.println(Arrays.deepToString(a));
    ```

Benefits of the Java Collections Framework

Reduces Programming Effort

The Collection framework provides data structures and algorithms, so that you can concentrate on only the important parts of your program rather than low-level "plumbing" required making it work.

Interoperability among unrelated APIs is another feature provided by the Java Collections Framework. So, you no more need to write adapter objects or conversion codes to connect APIs.

Increases Program Speed and Quality

High-performance, high-quality implementations of useful data structures and algorithms are some of the benefits provided by the Collections framework. Programs are easily tunable by switching collection implementations, as implementations of each interface are interchangeable. As you need not write your own data structures, you can concentrate on improving programs' efficiency.

Reduces Effort to Learn and to Use New APIs

Many APIs naturally take collections on input and furnish them as output. Previously, each such API had a small sub-API devoted to manipulating its collections. One had to learn each sub API just because of the little consistency among them. There were lots of chances of making mistakes when using them. But, with the advent of standard collection interfaces, the problem is solved.

Reduces Effort to Design New APIs

This is the flip side of the previous advantage. While creating an API that relies on collections, Designers and implementers don't have to reinvent the wheel each time. Instead, they can use standard collection interfaces.

Fosters Software Reuse

New data structures that conform to the standard collection interfaces are by nature reusable. The same goes for new algorithms that operate on objects that implement these interfaces.