```
public interface Collection
{
    //basic methods
    public boolean add(Object element);
    public boolean remove(Object element);
    public boolean contains(Object element);
    public void clear();
    public int size();
    public boolean isEmpty();

    //set methods
    public boolean addAll(Collection other);
    public boolean containsAll(Collection other);
    public boolean removeAll(Collection other);
    public boolean retainAll(Collection other);

    //misc methods
    public boolean equals(Object other);
    public int hashcode();
    public Iterator iterator();
    public Object[] toArray();
}
```

```
public interface Set
```

```
public interface List
{
    //index-based methods
    public void add(int index, Object element);
    public Object get(int index);
    public Object set(int index, Object element);
    public Object remove(int index);
    public int lastIndexOf(Object element);
    public List subList(int fromIndex, int toIndex);

    //basic methods
    public boolean add(Object element);
    public boolean remove(Object element);
    public boolean contains(Object element);
    public void clear();
    public int size();
    public boolean isEmpty();

    //set methods
    public boolean addAll(Collection other);
    public boolean addAll(int index, Collection other);
    public boolean containsAll(Collection other);
    public boolean removeAll(Collection other);
    public boolean retainAll(Collection other);

    //misc methods
    public boolean equals(Object other);
    public int hashcode();
    public Iterator iterator();
    public ListIterator listIterator();
    public ListIterator listIterator(int index);
    public Object[] toArray();
}
```

```
public interface Map
{
    //insertion and removal
    public Object put(Object key, Object value);
    public Object get(Object key);
    public Object remove(Object key);
    public void clear();

    //search
    public boolean containsKey(Object key);
    public boolean containsValue(Object value);
    public int size();
    public boolean isEmpty();

    //traversal
    public Set keyset();
    public Collection values();

    //misc
    public boolean equals(Object other);
    public int hashcode();
}
```

```
//create a new object of type T (this does not work!)
T newObject = new T();

public class MyClass<E implements Printable> {
    //generic type E also uses the interface Printable.
}

public class Drawing<V extends Shape>{
    //generic type V has Shape as a parent class
}
```

```
public class DataStorage<T> {
    private T data;

    public DataStorage(T data) {
        this.data = data;
    }

    public T getData(){
        return data;
    }

    public T setData(T data) {
        this.data = data;
    }
```

True a class can have more than one generic type

```
public class Book implements Comparable<T>
{
    private String author;
    private String title;
    private int pageCount;

public int compareTo(Book other) {
 int authorSort = this.author.compareTo
 (other.author);
 if (authorSort == 0) {
  return this.title.compareTo(other.title);
 } else {
  return authorSort;
 }
}
```

```
algorithm indexOf(A[0...n-1], value)
//input: An ARRAY A of integers, an int value that might be in the array
//output: An int that is the index of a specified int in the ARRAY A

for i <-- 0 to n-1
if  A[i] = value
return i
return -1
```

```
algorithm printVowels(A[0...n-1])
//input: ARRAY A of letters
//output: prints vowels, no return value

vowels <-- [a, e, i, o, u]

for i <-- 1 to n - 1
if vowels.contains( A[i])
print  A[i]
```

```
algorithm Intersection(A[0...n-1], X)
//input: A List A of unique strings (no duplicates)
//input: A List B of unique strings (no duplicates)
//output: an array listing strings that were in both A and B

t1[] <-- new ARRAY

for i <-- 1 to n - 1
for j <-- 1 to n - 1
if A[i] = B[j]
t1.add(A[i])
return t1
```

```
algorithm Find X(A[0...n-1], X)
//input: A sorted List A of integers
//input: An integer X
//output: Boolean of whether X is found in the array

Low <-- 0
High <-- n-1

while Low <= High
SearchIndex <-- (High + Low) / 2
CompareValue <-- A[SearchIndex]

if CompareValue = X
return true

if CompareValue < X
High <-- SearchIndex -1

if CompareValue > X
Low <-- SearchIndex + 1

return false
```

Generics were introduced to the Java language to provide tighter type checks at compile time and to support generic programming. To implement generics, the Java compiler applies **type erasure** to:
Replace all type parameters in generic types with their bounds or `Object` if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.
Insert type casts if necessary to preserve type safety.
Generate bridge methods to preserve polymorphism in extended generic types.
Type erasure ensures that no new classes are created for parameterized types; consequently, generics incur no runtime overhead.

When you declare the class and say <T extends Shape> or <T implements Shape> It replaces T with Shape. If you didn't say extends or implements it will be replaced with object.

Autoboxing is putting it in wrapper
Unboxing is returning it to primitive

f(n) = Ω(g(n)) – Omega is the tightest lower bound

f(n) = Θ(g(n)) – Theta is the tightest upper bound

Load factor is the percentage full

Primitive	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

compareTo() in Comparable & compare() in Comparator

Comparator is its own class and allows several sorts to happen sequentially and you can use separately in code segments

O() Best case scenario –
When you know you will be using it in that scenario like making a Stack or Queue with a List or using Bubble sort on something almost sorted

O() Average case scenario
If only rarely will it need to do the Worst Case Scenario it can help you decide whether to use one algorithm that has a better worst case scenario but on average the scenario will run significantly longer than the algorithm you are proposing.

O() Worst case scenario
What is the maximum amount of work it will take to run the algorithm? What if you do a search and the element isn't in the List, or you need to add at the beginning of an array. You need to know how it compares to a different algorithm to make a decision on which one to implement for your current use.

It is about knowing your code. What data you have and what your use case is trying to do . You use the Best algorithm for your case. Use the one that works best for your data, for your situation.

Binary search algorithm stack overflow error? No! Log of 1 million is only around 20 steps, so unless you have petabytes of information it would be hard to overflow.

Repeated halving principle: the number of times we can reduce a set of n elements by 1/2 before only one element remains is roughly $\log_2 n$

What advantage does interpolation search have over binary search? It can find the index of the value at log(logn) time (almost constant time).

What disadvantages? If the data is not increasing at a consistent slope it is unable to accurately determine the index of the element you are searching.

Operation	ArrayList	Singly Linked List	Double Linked List
add(element)	1	1	1
add(index, element)	n	n	n
addFirst(element)	n	1	1
addSecond(element)	n	1	1
remove(element), remove(index)	n	n	n
removeFirst(element)	n	1	1
removeLast(element)	1	n	1
contains(element)	n	n	n
get(index), set(index)	1	n	n

Low Load Factor Hash Table
Insert(x): -- O(1)
remove(x): --O(1)
contains(x): -- O(1)
size(): -- O(1)

High Load Factor Hash Table
Insert(x): -- O(n)
remove(x): --O(n)
contains(x): -- O(n)
size(): -- O(1)

Linear search -- Best – O(1), Worst – O(n), Avg – O(n)
Binary search -- Best – O(1), Worst – O(logn), Avg – O(logn) Interpolation search -- Best – O(1), Worst – O(n?), Avg – O(log(logn))

amortized running time—On average

Hash Code	A number or string that is used to represent an object. This value represents a summarization of the object and is often called a "digest."
Hash Table	An array that stores elements using Hash Codes.
Collision	This occurs when two elements would be placed in the same location in a Hash Table, given their hash codes.
Load Factor	The percent of used spots in a Hash Table which, if exceeded, results in a resize of the Hash Table.
Linear/Quadratic Probing	A collision resolution strategy where we look to adjacent spots in a Hash Table if a collision occurs.
Chaining	A collision resolution strategy where linked lists are used to hold elements at every position in a Hash Table. When a collision occurs, the new element is added to a linked list at the position determined by the element's hash code.

Binary Search Trees
Preorder: NLR
Inorder: LNR
Postorder: LRN
Breadth-first—queue

"if you override equals(), then you must also override hashcode()."

If you override .equals your hashes could go in different buckets and the map won't realize and keep both values.

```
public Iterator<Object> iterator() {
   return new BagIterator(data,this);
}
// inner classes
private static class BagIterator implements Iterator<Object>
{
   private Object[] outerClassData;
   private int currentPosition = 0;
   private Bag parentBag;
   private int currentModCount;

   public BagIterator(Object[] outerClassData, Bag parentBag)
   {
      this.outerClassData = outerClassData;
      this.parentBag = parentBag;
      //save the current state of the bag
      currentModCount = parentBag.modCount;
   }
   @Override
   public boolean hasNext()
   {
      //ensure that the data hasn't changed while we are iterating
      if (parentBag.modCount != currentModCount)
      {
         throw new ConcurrentModificationException (
               "You cannot change a bag while iterating over it!");
      }
      // make sure we still have a valid index and the current
      //element is not empty (null)
      return currentPosition < outerClassData.length &&
            outerClassData[currentPosition] != null;
   }
```

```
@Override
   public Object next()
   {
      //ensure that the data hasn't
changed while we are iterating
      if (parentBag.modCount != cur-
rentModCount)
      {
         throw new ConcurrentModifi-
cationException (
               "You cannot change a
bag while iterating over it!");
      }
      // return the current element,
and we need to increment
      //currentPosition so that it
points to the next element
      Object nextElement = outer-
ClassData[currentPosition];
      currentPosition++;
      return nextElement;
   }
```