Java provides a built-in interface, Comparable<T>, that is designed to allow programmers to compare objects with each other. The interface definition — which you do not have to supply because it is a built-in part of the language — declares one abstract method that must be defined in all concrete classes that implement the interface:

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

The T that appears between angle brackets in the name of this interface is a *formal type parameter*, just like the E that appears in the name of the [ArrayList<E>](https://www.eimacs.com/eimacs/mainpage?epid=E2557809216&cid=162149) class. Each time we replace T by the name of a [reference data type](https://www.eimacs.com/eimacs/mainpage?epid=E2375531766&cid=162149), we get an interface that prescribes a compareTo method that takes an argument of the data type in question. For example, a class implementing the Comparable<Person> interface must provide a method with return data type int and signature

compareTo( Person p ),

whereas a class implementing the Comparable<Building> interface must provide a method with return data type int and signature

compareTo( Building b ).

**Exercise 159**

A DateTime class implements the Comparable<DateTime> interface. What is the signature of its compareTo method?

compareTo( DateTime d )  
[You may use any variable in place of d.]

**Exercise 160**

The following code defines an Item class that implements the Comparable<Item> interface.

public class Item implements Comparable<Item>   
{   
  // define the compareTo method here

  public int compareTo( Item i )

   {

     return 0;

   }

}   
  
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    Item itemA = new Item(),   
         itemB = new Item();

System.out.println( itemA.compareTo( itemB ) );

  }   
}

Since the Item class implements the Comparable<Item> interface, we must give this class an appropriate compareTo method. In the area provided, define a compareTo method for the Item class that always returns the integer 0. (Pay close attention that you use the correct return data type and signature for this method.)

By modifying the main method as necessary and clicking the **Run** button, check that the integer 0 is indeed returned when you use your compareTo method to compare the Item itemA to the Item itemB, when you compare itemB with itemA, and when you compare each of these two Items with themselves.

The compareTo methods we asked you to write in Exercises [160](https://www.eimacs.com/eimacs/mainpage?epid=E2094883250&cid=162149#Exe145b) and [161](https://www.eimacs.com/eimacs/mainpage?epid=E2163743848&cid=162149#Exe145c) are worthless when it comes to the real purpose for which a compareTo method is required by the Comparable<T> interface. The reason why Java provides such an interface and insists that classes implementing the interface shall have a compareTo method is so that programmers would be able to arrange instances of such classes in order — to sort them. By returning the same result in many cases, the compareTo methods from Exercises 160 and 161 are of no help at all in helping us to decide which of any two Item objects "comes before" the other in any sense whatsoever.

To give you a feel for the kind of sorting ability it is useful to have, we make use of two static methods that are not in the Advanced Placement Java Subset: the Arrays.sort method and the Collections.sort method. Experiment with the following two code areas, editing the data, clicking the **Run** button, and watching what happens.

public class MainClass   
{   
  public static void printStringArray( String[] sArr )   
  {   
    for ( int i = 0 ; i < sArr.length ; i++ )   
    {   
      System.out.print( sArr[ i ] );   
      if ( i + 1 < sArr.length )   
        System.out.print( ", " );   
    }   
  }   
  
  public static void main( String[] args )   
  {

String[] array =

   { "spa", "despair", "spade", "vespa", "Espana",

     "spandex", "Sparta", "asparagus" };

    Arrays.sort( array );   
    printStringArray( array );   
  }   
}

Espana, Sparta, asparagus, despair, spa, spade, spandex, vespa

public class MainClass   
{   
  public static ArrayList<Integer> makeIntegerArrayList( int[] iArr )   
  {   
    ArrayList<Integer> a = new ArrayList<Integer>();   
    for ( int i = 0 ; i < iArr.length ; i++ )   
      a.add( iArr[ i ] );   
  
    return a;   
  }   
  
  public static void main( String[] args )   
  {

int[] array = { -1, 1, 12, -21, 1001, 203, -12 };

    ArrayList<Integer> aList = makeIntegerArrayList( array );   
    Collections.sort( aList );   
    System.out.println( aList );   
  }   
}

[-21, -12, -1, 1, 12, 203, 1001]

We reiterate that, for the AP Computer Science examination, you are not expected to understand, nor even to know about, the Arrays.sort or the Collections.sort methods.

These sorting methods work by comparing objects two by two, behind the scenes, and changing their positions in the sequence if any is found to be out of order. The compareTo method required by the Comparable<T> is intended to be such that it can be relied upon to perform such two-by-two comparisons in a way that is conducive to this kind of sorting operation. In fact, we have already seen a [compareTo](https://www.eimacs.com/eimacs/mainpage?epid=E2375527788&cid=162149#CompareTo) method that operates in exactly this fashion as far as Strings are concerned. Recall that:

If a and b are Strings, then the value of a.compareTo( b ) is

* a **negative** integer if and only if a precedes b in the [extended lexicographical order](https://www.eimacs.com/eimacs/mainpage?epid=E1961788402&cid=162149#LexOrder);
* the integer **zero** if and only if a and b are the same string;
* a **positive** integer if and only if b precedes a in the extended lexicographical order.

On the last two pages, we have been dealing with compareTo methods that are built into the Java language. How should we go about defining a compareTo method for ourselves that would be useful for the kind of two-by-two comparisons that contribute to a sorting operation? Let's look at an example.

#### Exercise 164

The TestClass class defined below has two instance variables (myName and myID) by reference to which we can order instances of the class. To get our discussion started, we provide a proposed compareTo method (which, as the exercise proceeds, will turn out to be inadequate).

public class TestClass implements Comparable<TestClass>   
{   
  private String myName;   
  private int myID;   
  
  public TestClass( String name, int id )   
  {   
    myName = name;   
    myID = id;   
  }   
  
  public String getName()   
  {   
    return myName;   
  }   
  
  public int getID()   
  {   
    return myID;   
  }

public int compareTo( TestClass t )

{

  if ( myID == t.getID() )

    return 0;

  return myName.compareTo( t.getName() );

}

}   
  
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    TestClass c1 = new TestClass( "zulima", 1 );   
    TestClass c2 = new TestClass( "fred", 2 );   
    TestClass c3 = new TestClass( "eric", 3 );   
    TestClass c4 = new TestClass( "jane", 2 );   
    TestClass c5 = new TestClass( "hermione", 1 );

System.out.println( c1.compareTo( c2 ) );

  }   
}

In the Advanced Placement examination, you will not be tested on the Comparable<T> interface. Instead, examination questions will use a simplification of this interface called "the *raw* Comparable interface". This is defined as follows:

public interface Comparable   
{   
  int compareTo( Object obj );   
}

There are two differences between this and the corresponding [definition](https://www.eimacs.com/eimacs/mainpage?epid=E2222767175&cid=162149#CompTDef) of the Comparable<T> interface:

* In the header there is no "<T>" to provide the link to the required data type of the argument of the required compareTo method.
* The argument of the required compareTo method is specified to be an Object.

Practically speaking, the raw Comparable interface behaves as if it were Comparable<Object>. The use of the raw Comparable interface instead of the type-specifying Comparable<T> interface complicates the definition of the required compareTo method by making it necessary to cast its argument to the desired data type, thereby raising the possibility of a *ClassCastException* being thrown. To illustrate the different style of compareTo definition used in the context of the raw Comparable interface, the following exercise revisits the TestClass class that was introduced in [Exercise 164](https://www.eimacs.com/eimacs/mainpage?epid=E2222767243&cid=162149#Exe146b).

**Exercise 165**

The following code redefines the TestClass class from Exercise 164, declaring in this case that it implements the raw Comparable interface. The compareTo method's definition is based on the revised version that resulted from part (c) of Exercise 164. Notice in particular the casting that occurs in the first statement in the body of its definition.

public class TestClass implements Comparable   
{   
  private String myName;   
  private int myID;   
  
  public TestClass( String name, int id )   
  {   
    myName = name;   
    myID = id;   
  }   
  
  public String getName()   
  {   
    return myName;   
  }   
  
  public int getID()   
  {   
    return myID;   
  }   
  
  public int compareTo( Object obj )   
  {   
    TestClass t = (TestClass)obj;   
    if ( myID == t.getID() )   
      return myName.compareTo( t.getName() );   
  
    return myID - t.getID();   
  }   
}   
  
public class MainClass   
{   
  public static void main( String[] args )   
  {   
    TestClass c1 = new TestClass( "zulima", 1 );   
    TestClass c2 = new TestClass( "fred", 2 );   
    TestClass c3 = new TestClass( "eric", 3 );   
    TestClass c4 = new TestClass( "jane", 2 );   
    TestClass c5 = new TestClass( "hermione", 1 );

System.out.println( c2.compareTo( c1 ) );

  }   
}

1

Java provides the Comparable<T> interface so as to make it possible for Java programmers to sort objects into order. The desire or need to sort things into order only usually arises when we are dealing with a collection of objects for which there is general agreement concerning the criterion (or criteria) to be used in order to perform that ordering. For example, it is generally agreed that

* if we want to sort a collection of Integers, then we may do so on the basis of the numerical values of the Integers in question;
* if we want to sort a collection of Doubles, then we may do so on the basis of the numerical values of the Doubles in question;
* if we want to sort a collection of Strings, then we may do so by treating the Strings in question as words and sorting them as if we were placing them in a dictionary.

In the case of the TestClass objects in the last two exercises, the ordering criterion we ended up using is a natural one: first, sort on the basis of the numerical values of the myID variables and then, if those values happen to be equal, sort on the basis of the lexicographical ordering of the myName variable. This is very similar to a common ordering criterion for databases (such as those maintained by your school) in which the computer record concerning you has fields for your first name and your last name. Such records can be sorted, first, on the basis of last name and then, if two or more students have the same last name, on the basis of first name.

Numerical order, lexicographical order, and other orders based on date or time or age or time to run a 40-meter dash or on some combination of these are said to be *natural* orders. They are suggested by the very nature of the objects we want to place in order or by the nature of one or more of the characteristics of those objects. The purpose of the compareTo method required by the Comparable<T> interface is to provide programmatic access to the natural ordering of T objects.

**The Prerequisite for Comparability**

If T is a class (either built into Java or defined by a programmer), then in order for it to be sensible to think in terms of implementing a Comparable<T> interface, it must be the case that there is a *natural ordering* among T objects. For the Advanced Placement exam you are not required to know the technical requirements for such a natural ordering, but if you are interested in knowing what these are, click the *Show »* link below.

*Technical requirements*

« Hide

* If a and b are T objects, and if a precedes b in the natural ordering, then b does *not* precede a in that ordering;
* If a, b, and c are T objects, and if a precedes b and b precedes c in the natural ordering, then a precedes c in that ordering; and
* If a and b are T objects such that neither one precedes the other in the natural ordering, then, for any T object c,
  + if a precedes c, then b also precedes c; and
  + if c precedes a, then c also precedes b.

**Implementing Comparability**

If T is a class such that there is a natural ordering of the above kind among T objects, then T is a good candidate for implementing the Comparable<T> interface. Of course, if T is not a built-in Java class and we declare that T implements the Comparable<T> interface, then we are giving ourselves the responsibility of defining a compareTo method for the T class. At this point, Java imposes upon us what is called a *general contract* for the compareTo method. The purpose of this contract is to remind the programmer that programmatic comparisons of T are not going to work correctly unless the compareTo method is defined in such a way that it faithfully reflects the natural ordering that exists among T objects. (Each of the built-in classes, such as String and Integer, that implement a corresponding Comparable interface has a built-in compareTo method that already abides by this general contract. We only have to think about the contract when defining classes of our own.) For the Advanced Placement exam you are not required to know the technical details of the general contract for compareTo, but if you are interested in knowing what these are, click the *Show »* link below.

*Technical requirements*

« Hide

If t is an instance of T and x is either an instance of T or is automatically cast to an instance of T or is automatically wrapped to an instance of T, then t.compareTo( x ) must

* return a negative integer if t precedes x (or whatever results from the automatic modification of x) in the natural ordering of T objects;
* return the integer zero if t and x (or whatever results from the automatic modification of x) are equal according to the natural ordering of T objects;
* return a positive integer if x (or whatever results from the automatic modification of x) precedes t in the natural ordering of T objects.

In fact, Java imposes a similar *general contract* on every method declared in its built-in interfaces (of which the Comparable<T> interface is one). Each such general contract is a verbal description that governs how any implementation of the method is required to behave.