GENERICS AND ARRAY-BASED LISTS

TOPICS

OUTLINE

---

1. Interface: `Cloneable`

- **Reminder:** inheritance, interfaces, class `Object`.
- The method `clone` of the class `Object` is a protected method inherited by every class in Java. The method clone cannot be invoked by an object outside the definition of its class.
- It provides a bit-by-bit copy of the objects data in storage (shallow copy of object's data) --> In order to make a deep copy of an objects data, its class must override the clone method from class `Object`.
- **NOTE:** The interface `Cloneable` has no method headings that need to be implemented --> **Classes that implement this interface must only redefine the `clone()` method.**
- **Example:**

```
public class Time implements Cloneable {...}
```

- When writing the clone method, follow these steps:
    - Invoke the `clone` method of the superclass
    - Change the values of instance variables of mutable types
- **NOTE:** The method `clone` of the class `Object` throws `CloneNotSupportedException`
- **Example:** Same method `clone()` in class `Time, Date, Person`

```
public Object clone() {
    try  {
        return super.clone();
    }
    catch (CloneNotSupportedException e) {
        return null;
    }
}
```

- **NOTE:** The method `clone` returns a reference of the type Object or the value `null` --> must typecast the returned object to a reference of the same type as the class you work with.
- **Example:**

```
Time t1 = new Time(11, 12, 13);
Time t2 = (Time) t1.clone();
```

- If the class uses composition (has instance variables of type class), then the clone method has to

change the values of those instance variables.

- **Example:** method `clone()` in class `PersonalInfo`

```
public Object clone() {
    try {
        PersonalInfo copy = (PersonalInfo) super.clone();
        copy.birthDate = (Date) birthDate.clone();
        copy.fullName = (Person) fullName.clone();
        return copy;
    }
    catch (CloneNotSupportedException e) {
        return null;
    }
}
```

---

2. Interface: `Comparable`

- The interface `Comparable` has only one method heading, which is `compareTo` (not a member of `Object`) --> Used to force a class to provide an appropriate definition of the method `compareTo`. It provides a means of fully ordering objects.
- Values of two objects of the same class can be properly compared (`equals` compares only for equality) Many types have the notion of a natural ordering that describes whether one value of that type is "less than" or "greater than" another: numeric values, strings (lexical/alphabetical order), times, dates, etc. Not all types have a natural ordering (ex: `Point`)
- **NOTE:** The interface `Comparable` has no method headings that need to be implemented --> **Classes that implement this interface must only redefine the `compareTo()` method.**
- The method `compareTo` compares this object with the object passed for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the object passed.
- **Example:**

```
public class Time implements Comparable {...}
```

- **NOTE:** If a class implements multiple interfaces, separate all interfaces names using commas.
- **Example:**

```
public class Time implements Cloneable, Comparable {...}
```

- **Example:** method `compareTo()` in class `Time`:

```
public int compareTo(Object othertime) {
    Time temp = (Time) othertime;
    int hrDiff = hrs - temp.hrs;
    if (hrDiff != 0)
        return hrDiff;
    int minDiff = mins - temp.mins;
    if (minDiff != 0)
        return minDiff;
    return secs - temp.secs;
}
```

- **Example:** method `compareTo()` in class `Date`:

```
public int compareTo(Object otherDate) {
    Date temp = (Date) otherDate;
    int yearDiff = year - temp.year;
    if (yearDiff != 0)
        return yearDiff;
    int monthDiff = month - temp.month;
```

```
    if (monthDiff != 0)
        return monthDiff;
    return day - temp.day;
}
```

- **Example:** method `compareTo()` in class `Person`:

```
public int compareTo(Object otherPerson) {
    Person temp = (Person) otherPerson;
    int compare = lastName.compareTo(temp.lastName);
    if(compare == 0)
        compare = firstName.compareTo(temp.firstName);
    return compare;
}
```

- **Example:** method `compareTo()` in class `PersonalInfo`:

```
public int compareTo(Object other) {
    PersonalInfo temp = (PersonalInfo) other;
    int compare = personID - temp.personID;
    if(compare == 0)
        compare = fullName.compareTo(temp.fullName);
    if(compare == 0)
        compare = birthDate.compareTo(temp.birthDate);
    return compare;
}
```

- Check here the (complete) new definition for class `Time`. (including `clone()` and `compareTo()`)
- Check here the (complete) new definition for class `Date`. (including `clone()` and `compareTo()`)
- Check here the (complete) new definition for class `Person`. (including `clone()` and `compareTo()`)
- Check here the (complete) new definition for class `PersonalInfo`. (including `clone()` and `compareTo()`)

---

3. Primitive Data Types and Wrapper Classes

- **Definition: Wrapper Classes** = Classes `Integer, Double, Character, Long, Float, Boolean`, etc, provided so that values of primitive data types can be treated as objects.
- A wrapper is an object whose sole purpose is to hold a primitive value.

Primitive type	Wrapper class
`int`	`Integer`
`double`	`Double`
`float`	`Float`
`char`	`Character`
`boolean`	`Boolean`
`long`	`Long`

- The `Integer` class wraps a value of the primitive type `int` in an object. An object of type `Integer` contains a single instance variable whose type is `int`. The class provides several methods for converting an `int` to a `String` and a `String` to an `int`, as well as other methods useful when dealing with an `int`. Take a look at some methods in class `Integer`
- **Definition:** The conversion between the primitive type and the wrapper class = **boxing and unboxing.**

- As of Java Standard Edition 5.0, Java **automatically** converts values back and forth between the primitive type and the corresponding wrapper class = **auto-boxing and auto-unboxing.**
- **How auto-boxing works:** if an `int` is passed in a place where an `Integer` is required, the compiler will make a (behind the scenes) call to the wrapper class constructor (`Integer`).
- **How auto-unboxing works:** if an `Integer` is passed in a place where an `int` is required, the compiler will make a (behind the scenes) call to the `intValue` method.
- Similar behavior takes place for the 7 other primitive types/wrapper classes pairs.
- **Example:**

Auto-boxing (of the int type)	Auto-unboxing (of the Integer type)

`int x;`
`Integer num;`
 	
`num = 10; //auto boxing`
 	
`Equivalent to the statement:`
`num = new Integer(10);`
	`int x;`
	`Integer num;`
	
	`x = num; //auto-unboxing`
	
	`Equivalent to the statement:`
	`x = num.intValue();`

- **When to use auto-boxing and auto-unboxing?** Only when there is a mismatch between reference types and primitives (for example, when you have to put numerical values into a collection - to be discussed later). It is not appropriate to use them for scientific computing, or other performance-sensitive numerical code. An `Integer` is not a perfect substitute for an `int`; auto-boxing and unboxing blur the distinction between primitive types and reference types, but they do not eliminate it (for example, the wrapper classes have limitations - cannot change the value stored in an object). Although boxing and unboxing can be quite convenient, this feature can generate confusion and should be used with care.

---

4. Generics

- One important goal of OOP is to provide the ability to write reusable, generalized code. One important mechanism that supports this goal = generics. **The idea:** if the implementation is identical except for the basic type of the object --> use a generic implementation to describe the basic functionality. **Example:** When sorting an array, the logic in bubble sort is independent of the types of objects being sorted. Same for searching, inserting, deleting, etc.
- Java 5 supports generic methods and generic classes. Writing generic classes requires more work - the constructs of the language are quite complex (and sometimes tricky!)
- **Definition: Generics** = powerful means of writing generalized code that can be used by any class in any hierarchy represented by the **type parameter**. Class definitions that include a type parameter are called generic types.
- **Definition: Type parameter** = identifier that specifies a generic type name. Any non-keyword identifier can be used for the type parameter, but by convention, the parameter starts with an uppercase letter. The type parameter can be used like other types used in the definition of a class, but cannot represent primitive types (see the note below). Also known as type variables - used to:
- Declare the return type of the method
- Declare formal parameters
- Declare local variables
- **NOTE:** A type parameter cannot be used everywhere a type name can be used. In particular, the type parameter cannot be used in simple expressions using `new` to create a new object. **Example:** `T someObject = new T();`
`T[] someArray = new T[SIZE];` --> Compiling ERROR!

- **NOTE:** A primitive type cannot be plugged in for a type parameter --> the type plugged in for a type parameter must always be a reference type. However, Java has automatic boxing so, for wrapper classes this is not a big restriction (use wrapper classes instead of primitive data types!). Also, reference types can include arrays.
- **Definition: Generic method** = Method defined using type parameter. The type list precedes the return type. The type is used as the return type, OR the type is used in more than one parameter, OR the type is used to declare a local variable. The general syntax for a generic method is: **`modifier(s) <T> methodType methodName(formal parameters) {...}`**
- **Examples:**

```
//generic method to print array
public static <T> void print(T[] list) {
    for(int i = 0; i < list.length; i++)
        System.out.print(list[i] + " ");
    System.out.println();
}
```

Calls:

```
Integer[] intList = {1, 2, 3, 4, 5};
Double[] doubleList = {1.1, 2.2, 3.3};
String[] strList = {"List", "Stack", "Queue", "Tree"};

print(intList);
print(numList);
print(strList);
```

```
//generic method for linear search
public static <T> int seqSearch(T[] list, T searchItem) {
    for(int i = 0; i < list.length; i++)
        if(list[i].equals(searchItem))
            return i;
    return -1;
}
```

- Generic methods and **bounded type parameters** --> There are situations when the type parameter `<T>` must be restricted. For example: we plan to design a method to find the larger value of two objects. The method should work with built-in as well as user-defined classes. Objects are compared using the method `compareTo` --> the method should work only with classes that provide a definition of the method `compareTo`.
- **Examples:**

```
//findMax/2 objects
public static <T extends Comparable<T> > T findMax(T x, T y) {
    return x.compareTo(y) >= 0? x: y;
}

//findMax/array
public static <T extends Comparable<T> > T findMax(T[] list) {
    int maxIndex = 0;
    for(int i = 0; i < list.length; i++)
        if(list[i].compareTo(list[maxIndex]) > 0)
            maxIndex = i;
    return list[maxIndex];
```

```
    }

    //bubble sort
    public static <T extends Comparable<T>> void bubbleSort(T[] list) {
        for (int pass = 0; pass < length - 1; pass++) {
             for (int i = 0; i < length - 1 - i; i++) {
                 Comparable<T> listElem = (Comparable<T>) list[i];
                 if (listElem.compareTo(list[i + 1]) > 0) {
                     T temp = list[i];
                     list[i] = list[i + 1];
                     list[i + 1] = temp;
                 }
             }
         }
    }
```

- **NOTE:** Always use the keyword `extends` regardless of whether the type parameter extends a class or an interface
- **NOTE:** If a type parameter is bounded by more than one class (or interface) class names are separated using the symbol &
- **Definition: Generic class (parameterized or parametric class)** = a class that is defined with a parameter for a type. The type parameter is included in angular brackets < > after the class name in the class definition heading. The general syntax for a generic class is: **`modifier(s) className <T> modifier(s) {...}`**
- Generic classes are used to write a single definition for a set of related classes.
- **NOTE:** A class definition with a type parameter is stored in a file and compiled just like any other class. Once a parameterized class is compiled, it can be used like any other class. However, the class type plugged in for the type parameter must be specified before it can be used in a program.
- **NOTE:** An instantiation of a generic class cannot be an array base type --> use an `ArrayList` instead.
- **NOTE:** A generic class definition can have any number of type parameters --> multiple type parameters are listed in angular brackets just as in the single type parameter case, but are separated by commas.

---

5. Abstract Data Types (ADT)

- **Definition: Abstract Data Type (ADT)** = A data type that specifies the logical properties without the implementation details (information hiding --> the ADT hides the implementation details from the programs using the ADT --> the users can use the operations of an ADT without knowing how they have been implemented).
- ADT = a collection of data and a set of operations on that data (while a data structure is a construct within a programming language that stores a collection of data). Describes what the collection does, not how it does it.
- Data abstraction is the result of ADT operations between data structures and the program that accesses the data within these data structures. The goal of data abstraction is to separate the operations on data from the implementation of these operations. A program should not depend on the details of an ADT implementation because ADT's properties (domain and operations) are specified independently of any particular implementation.
- An ADT separates the logical properties of a data type (what are the possible values, what operations will be needed) from its implementation. In other words, data abstraction separates the qualities of an object from the details of how it works.

- **ADT properties:**
- Set of values (domain)
- Allowable operations on those values.

---

6. Array-Based Lists

- **Definitions:**
- **List** = Linear data structure whose components could be accessed sequentially, one after the other. A list is a collection of elements of the same type.
- **Components of a list** = List elements OR list items.
- **Head (front)** = First element of the list.
- **Tail (back, end)** = Last element of the list.
- **Length of a list** = The number of elements in the list. Conceptually, there is no upper bound for length, but a computer's memory size is bounded --> "list is full" means the list has reached maximum length.
- **List properties:**
- The length of the list could change over time, as new items are inserted and old items are deleted (varying length).
- All list elements have the same type (homogeneous components).
- Access to list elements is sequential (through an implicit list cursor --> a pointer that keeps track of the location, while going in a sequence through the list).
- **Linear relationship -->** Each element except the head has a unique predecessor, and each element except the tail has a unique successor.
- **Unordered/unsorted list** = A list in which data items are placed in no particular order; the only relationship between data elements is the list predecessor and successor relationships.
- **Key** = The attribute used to determine the logical/physical order of the list elements.
- **Ordered/sorted list** = A list in which the list elements are in an order that is sorted in some way -- either numerically or alphabetically by the elements themselves, or by a component (key). A sorted list is a list that is sorted by the value in the key; there is a semantic relationship among the keys of the items in the list. If a list cannot contain elements with duplicate keys, it is said to have unique keys.
- **Common operations performed on a list**

    - Create the list
    - Determine whether the list is empty or full
    - Find the size of the list
    - Destroy (or clear) the list
    - Make a copy of the list
    - Print the list
    - Insert an item at the specified location
    - Remove an item at the specified location
    - Replace an item at the specified location
    - Retrieve an item at the specified location
    - Search the list for a given item
- **Advantage of using ADT Lists over arrays:** Much easier to perform insertions and deletions (less operations).
- **Disadvantage of using ADT Lists over arrays:** Direct access of a component is impossible (advance first to the element with index i, then print; with arrays could just print the element with index i).
- We prefere using a list ADT when the frequency of inserting and deleting elements is significantly

greater than the frequency of selecting, storing or retrieving existing elements.

- To implement the **List ADT**, we must
  1) Choose a concrete data representation for the list. An effective, convenient, and common way to process a list is to store it in an array. Initially the size of the array is larger than the size of the list; the list can grow to a larger size. We must know how full the array is.
  2) Implement the list operations
- Variables needed to maintain and process the list in an array
  - The array, `list`, holding the list elements
  - A variable, `length`, to store the length of the list
  - A variable, `maxSize`, to store the size of the array
- The list can be sorted/ordered or unsorted/unordered - however, the algorithms to implement certain operations are the same. Common operations for the sorted/unsorted list are implemented in class `ArrayListClass`. The `ArrayListClass` is an abstract class (we do not want to instantiate objects of this class) - it does not implement all the operations of the interface `ArrayListADT`.

```
//Interface: ArrayListADT
//extends Cloneable
public interface ArrayListADT<T> extends Cloneable {
    public boolean isEmpty(); //Method to determine whether the list is empty.
    public boolean isFull();  //Method to determine whether the list is full.
    public int listSize();    //Method to return the number of elements in the list.
    public int maxListSize(); //Method to return the maximum size of the list.
    public void print();      //Method to output the elements of the list.
    public Object clone();    //Returns a copy of objects data in store. Clones only
the references, not the objects
    public boolean isItemAtEqual(int location, T item); //Method to determine
whether item is the same as the item in the list at location.
    public void insertAt(int location, T insertItem);   //Method to insert
insertItem in the list at the position
    public void insertEnd(T insertItem); //Method to insert insertItem at the end of
the list.
    public void removeAt(int location);  //Method to remove the item from the list
at location.
    public T retrieveAt(int location);   //Method to retrieve the element from the
list at location.
    public void replaceAt(int location, T repItem); //Method to replace the element
in the list at location with repItem.
    public void clearList();  //Method to remove all the elements from the list.
    public int search(T searchItem);     //Method to determine whether searchItem is
in the list.
    public void remove(T removeItem);    //Method to remove an item from the list.
}

//Class: ArrayListClass implements
//Interface: ArrayListADT
public abstract class ArrayListClass<T> implements ArrayListADT<T>, Cloneable {
    protected int length;  //to store the length of the list
    protected int maxSize; //to store the maximum size of the list
    protected T[] list;    //array to hold the list elements

    //Default constructor
    public ArrayListClass() {
```

```
        maxSize = 100;
        length = 0;
        list = (T[]) new Object[maxSize];
    }

    //Alternate Constructor
    public ArrayListClass(int size) {
        if(size <= 0) {
            System.err.println("The array size must be positive. Creating an array
of size 100. ");
            maxSize = 100;
        }
        else
            maxSize = size;
        length = 0;
        list = (T[]) new Object[maxSize];
    }

    public boolean isEmpty() {
        return (length == 0);
    }

    public boolean isFull() {
        return (length == maxSize);
    }

    public int listSize() {
        return length;
    }

    public int maxListSize() {
        return maxSize;
    }

    public void print() {
        for (int i = 0; i < length; i++)
            System.out.print(list[i] + "  ");
        System.out.println();
    }

    public Object clone() {
        ArrayListClass<T> copy = null;
        try {
           copy = (ArrayListClass<T>) super.clone();
        }
        catch (CloneNotSupportedException e) {
            return null;
        }
        copy.list = (T[]) list.clone();
        return copy;
    }
```

```
    public boolean isItemAtEqual(int location, T item) {
        if (location < 0 || location >= length) {
            System.err.println("The location of the item to be compared is out of
range.");
            return false;
        }
        return (list[location].equals(item));
    }

    public void clearList() {
        for (int i = 0; i < length; i++)
            list[i] = null;
        length = 0;
        System.gc();   //invoke garbage collector
    }

    public void removeAt(int location) {
        if (location < 0 || location >= length)
            System.err.println("The location of the item to be removed is out of
range.");
        else {
            for(int i = location; i < length - 1; i++)
                 list[i] = list[i + 1];
            list[length - 1] = null;
            length--;
        }
    }

    public T retrieveAt(int location) {
        if (location < 0 || location >= length) {
            System.err.println("The location of the item to be retrieved is out of
range.");
            return null;
        }
        else
            return list[location];
    }

    public abstract void insertAt(int location, T insertItem);
    public abstract void insertEnd(T insertItem);
    public abstract void replaceAt(int location, T repItem);
    public abstract int search(T searchItem);
    public abstract void remove(T removeItem);
}
```

---

7. ADT: Unordered List

```
//Class: UnorderedArrayList extends
//Class: ArrayListClass
public class UnorderedArrayList<T> extends ArrayListClass<T> {
    //Default constructor
```

```
    public UnorderedArrayList() {
        super();
    }

    //Alternate Constructor
    public UnorderedArrayList(int size) {
        super(size);
    }

    //Bubble Sort
    public void bubbleSort() {
        for (int pass = 0; pass < length - 1; pass++) {
            for (int i = 0; i < length - 1; i++) {
               Comparable<T> listElem = (Comparable<T>) list[i];
                if (listElem.compareTo(list[i + 1]) > 0) {
                    T temp = list[i];
                    list[i] = list[i + 1];
                    list[i + 1] = temp;
                }
            }
        }
    }

    //implementation for abstract methods defined in ArrayListClass
    //unordered list --> linear search
    public int search(T searchItem) {
        for(int i = 0; i < length; i++)
            if(list[i].equals(searchItem))
                return i;
        return -1;
    }

    public void insertAt(int location, T insertItem) {
        if (location < 0 || location >= maxSize)
            System.err.println("The position of the item to be inserted is out of
range.");
        else if (length >= maxSize)
            System.err.println("Cannot insert in a full list.");
        else {
            for (int i = length; i > location; i--)
                list[i] = list[i - 1];  //shift right
            list[location] = insertItem;
            length++;
        }
    }

    public void insertEnd(T insertItem) {
        if (length >= maxSize)
            System.err.println("Cannot insert in a full list.");
        else {
            list[length] = insertItem;
            length++;
```

```
        }
    }

    public void replaceAt(int location, T repItem)  {
        if (location < 0 || location >= length)
            System.err.println("The location of the item to be replaced is out of
range.");
        else
            list[location] = repItem;
    }

    public void remove(T removeItem) {
        int i;
        if (length == 0)
            System.err.println("Cannot delete from an empty list.");
        else {
            i = search(removeItem);
            if (i != -1)
                removeAt(i);
            else
                System.out.println("Cannot delete! The item to be deleted is not in
the list.");
        }
    }
}

//Class: ClientUnorderedListInt
//Client program to test the UnorderedList with Integer objects
import java.util.Scanner;
public class ClientUnorderedListInt {
    public static final int SIZE = 20;
    public static void main(String[] args)  {
        Scanner input = new Scanner(System.in);
        UnorderedArrayList<Integer> list = new UnorderedArrayList<Integer>(SIZE);
        UnorderedArrayList<Integer> list2;
        Integer n;
        int index;
        list.insertEnd(200);
        list.insertEnd(15);
        list.insertEnd(3);
        list.insertEnd(10);
        list.insertEnd(100);
        list.insertEnd(37);
        System.out.println("Testing .insertEnd. Inserted in the list values: 200 15
3 10 100 37");
        System.out.println("The original list is: ");
        list.print();
        System.out.print("Testing .search. Enter value to search for: ");
        n = input.nextInt();
        index = list.search(n);
        if(index != -1)
            System.out.println("Found " + n + " in the element with index " +
```

```
index);
        else
            System.out.println(n + " not found in this list");
        System.out.print("Testing .remove. Enter value to remove from list: ");
        n = input.nextInt();
       list.remove(n);
        System.out.println("The list after removing " + n + " is: ");
        list.print();
        list2 = copyList(list);
        System.out.println("Testing client method copyList. The copy list (list2)
is: ");
        list2.print();
        n = list2.retrieveAt(0);
        System.out.println("Testing .retrieveAt. First element in list2 is: " + n);
        list2.replaceAt(0, 25);
        System.out.println("Testing .replaceAt. Changed the first element in list2
to 25. ");
        System.out.println("After change, list2 is: ");
        list2.print();
        System.out.println("Was the original list changed? If not, list2 is a deep
copy! The original list is: ");
        list.print();
        System.out.println("Testing .bubbleSort. The original list sortd is: ");
        list.bubbleSort();
        list.print();
    }

    public static UnorderedArrayList<Integer> copyList (UnorderedArrayList<Integer>
otherList) {
        UnorderedArrayList<Integer> tempList = new UnorderedArrayList<Integer>
(otherList.maxListSize());
        Integer n;
        for (int i = 0; i < otherList.listSize(); i++) {
            n =  otherList.retrieveAt(i);
            tempList.insertEnd(n);
        }
        return tempList;
    }
}
```

OUTPUT:

```
Testing .insertEnd. Inserted in the list values: 200 15 3 10 100 37
The original list is:
200  15  3  10  100  37
Testing .search. Enter value to search for: 100
Found 100 in the element with index 4
Testing .remove. Enter value to remove from list: 15
The list after removing 15 is:
200  3  10  100  37
Testing client method copyList. The copy list (list2) is:
200  3  10  100  37
Testing .retrieveAt. First element in list2 is: 200
```

```
Testing .replaceAt. Changed the first element in list2 to 25.
After change, list2 is:
25  3  10  100  37
Was the original list changed? If not, list2 is a deep copy! The original list is:
200  3  10  100  37
Testing .bubbleSort. The original list sortd is:
3  10  37  100  200
```

```
//Class: ClientUnorderedListTime
//Client program to test the UnorderedList with Time objects
public class ClientUnorderedListTime {
    public static final int SIZE = 20;
    public static void main(String[] args)  {
        UnorderedArrayList<Time> timeList = new UnorderedArrayList<Time>(SIZE);
        UnorderedArrayList<Time> temp1;
        UnorderedArrayList<Time> temp2;
        timeList.insertEnd(new Time(1, 2, 3));
        timeList.insertEnd(new Time(4, 5, 6));
        timeList.insertEnd(new Time(7, 8, 9));

        System.out.println("The original list is: ");
        timeList.print();
       temp1 = (UnorderedArrayList<Time>) timeList.clone();
        System.out.println("The copy list (temp1) after call to clone: ");
        temp1.print();
        temp2 = copyList(timeList);
        System.out.println("The copy list (temp2) after call to copyList: ");
        temp2.print();
        Time tl;
        tl = temp1.retrieveAt(0);
        tl.setTime(11, 12, 13);
        System.out.println("Changed the first element in temp1 to 11:12:13");
        System.out.println("After change, temp1 is: ");
        temp1.print();
        System.out.println("Was the original list changed? If yes, temp1 is a
shallow copy! The original list is: ");
        timeList.print();
        System.out.println("Was temp2 changed? If not, temp2 is a deep copy. temp2
is: ");
        temp2.print();
    }

    public static UnorderedArrayList<Time> copyList (UnorderedArrayList<Time>
otherList) {
        UnorderedArrayList<Time> tempList = new UnorderedArrayList<Time>
(otherList.maxListSize());
        Time tl;
        for (int i = 0; i < otherList.listSize(); i++) {
            tl =  otherList.retrieveAt(i);
            tempList.insertEnd((Time) tl.clone());
        }
        return tempList;
```

```
    }
}
```

OUTPUT:

```
The original list is:
01:02:03  04:05:06  07:08:09
The copy list (temp1) after call to clone:
01:02:03  04:05:06  07:08:09
The copy list (temp2) after call to copyList:
01:02:03  04:05:06  07:08:09
Changed the first element in temp1 to 11:12:13
After change, temp1 is:
11:12:13  04:05:06  07:08:09
Was the original list changed? If yes, temp1 is a shallow copy! The original list
is:
11:12:13  04:05:06  07:08:09
Was temp2 changed? If not, temp2 is a deep copy. temp2 is:
01:02:03  04:05:06  07:08:09
```

---

8. ADT: Ordered List

```
//Class: OrderedArrayList extends
//Class: ArrayListClass
public class OrderedArrayList<T> extends ArrayListClass<T>{
    //Default constructor
    public OrderedArrayList() {
        super();
    }

    //Alternate constructor
    public OrderedArrayList(int size) {
        super(size);
    }

    //implementation for abstract methods defined in ArrayListClass
    //ordered list --> binary search
    public int search(T item) {
        int first = 0;
        int last = length - 1;
        int middle = -1;

        while (first <= last)  {
            middle = (first + last) / 2;
            Comparable<T> listElem = (Comparable<T>) list[middle];
            if (listElem.compareTo(item) == 0)
                return middle;
            else
                if (listElem.compareTo(item) > 0)
                    last = middle - 1;
                else
                    first = middle + 1;
        }
```

```
        return -1;
    }

    public void insert(T item) {
        int loc;
        boolean found = false;
        if (length == 0)         //list is empty
            list[length++] = item;  //insert item and increment length
        else if (length == maxSize) //list is full
            System.err.println("Cannot insert in a full list.");
        else {
            for (loc = 0; loc < length; loc++) {
                Comparable<T> temp = (Comparable<T>) list[loc];
                if (temp.compareTo(item) >= 0) {
                    found = true;
                    break;
                }
            }
            //starting at the end, shift right
            for (int i = length; i > loc; i--)
                list[i] = list[i - 1];
            list[loc] = item;
            length++;
        }
    }

    /* Another version for insert:
    public void insert(T item) {
        int loc;
        boolean found = false;
        if (length == 0)            //list is empty
            list[length++] = item;  //insert item and increment length
        else if (length == maxSize) //list is full
            System.err.println("Cannot insert in a full list.");
        else {
            int i = length - 1;
            //while (i >= 0 && list[i] > item) {
            while (i >= 0 && ((Comparable<T>) list[i]).compareTo(item) > 0) {
                list[i + 1] = list[i];
                i--;
            }
            list[i + 1] = item;  // Insert item
            length++;
       }
    } */

    public void insertAt(int location, T item)  {
        if (location < 0 || location >= maxSize)
            System.err.println("The position of the item to be inserted is out of
range.");
        else if (length == maxSize)  //list is full
```

```
            System.err.println("Cannot insert in a full list.");
        else {
            System.out.println("This is a sorted list. Doing insert in place (call
to insert).");
            insert(item);
        }
    }

    public void insertEnd(T item) {
        if (length == maxSize)  //the list is full
            System.err.println("Cannot insert in a full list.");
        else {
            System.out.println("This is a sorted list. Doing insert in place (call
to insert).");
            insert(item);
        }
    }

    public void replaceAt(int location, T item) {
        //the list is sorted!
        //is actually removing the element at location and inserting item in place
        if (location < 0 || location >= length)
             System.err.println("The position of the item to be replaced is out of
range.");
        else {
            removeAt(location);//method in ArrayListClass
            insert(item);
        }
    }

     public void remove(T item) {
        int loc;
        if (length == 0)
            System.err.println("Cannot delete from an empty list.");
        else  {
            loc = search(item);
            if (loc != -1)
                removeAt(loc);//method in ArrayListClass
            else
                System.out.println("The item to be deleted is not in the list.");
        }
    }

    /*Another version for remove:
    public void remove(T item) {
        int loc;
        if (length == 0)
            System.err.println("Cannot delete from an empty list.");
        else  {
            loc = search(item);
            if (loc != -1) {
                for(int i = loc; i < length - 1; i++)
```

```
                    list[i] = list[i + 1]; //shift left
                length--;
            }
            else
                System.out.println("The item to be deleted is not in the list.");
        }
    } */
}

//Class: ClientUnorderedListInt
//Client program to test the UnorderedList with Integer objects
import java.util.Scanner;
public class ClientOrderedListInt {
    public static final int SIZE = 20;
    public static void main(String[] args)  {
        Scanner input = new Scanner(System.in);
        OrderedArrayList<Integer> list = new OrderedArrayList<Integer>(SIZE);
        OrderedArrayList<Integer> list2;
        Integer n;
        int index;
        list.insert(5);
        list.insert(3);
        list.insert(10);
        list.insert(100);
        list.insert(37);
        System.out.println("Testing .insert. Inserted in the list the values: 5 3 10
100 37");
        System.out.println("The original list is: ");
        list.print();
        System.out.print("Testing .search. Enter value to search for: ");
        n = input.nextInt();
        index = list.search(n);
        if(index != -1)
            System.out.println("Found " + n + " in the element with index " +
index);
        else
            System.out.println(n + " not found in this list");
        System.out.print("Testing .remove. Enter value to remove from list: ");
       n = input.nextInt();
        list.remove(n);
        System.out.println("The list after removing " + n + " is: ");
        list.print();

        list2 = copyList(list);
        System.out.println("Testing client method copyList. The copy list (list2)
is: ");
        list2.print();
        n = list2.retrieveAt(0);
        System.out.println("Testing .retrieveAt. First element in list2 is: " + n);
        list2.replaceAt(0, 25);
        System.out.println("Testing .replaceAt. Removed first element in list2,
```

```
added value 25. ");
        System.out.println("After change, list2 is: ");
        list2.print();
        System.out.println("Was the original list changed? If not, list2 is a deep
copy! The original list is: ");
        list.print();
    }

    public static OrderedArrayList<Integer> copyList (OrderedArrayList<Integer>
otherList) {
        OrderedArrayList<Integer> tempList = new OrderedArrayList<Integer>
(otherList.maxListSize());
        Integer n;
        for (int i = 0; i < otherList.listSize(); i++) {
            n =  otherList.retrieveAt(i);
            tempList.insert(n);
        }
        return tempList;
    }
}
```

OUTPUT:

```
Testing .insert. Inserted in the list the values: 5 3 10 100 37
The original list is:
3  5  10  37  100
Testing .search. Enter value to search for: 37
Found 37 in the element with index 3
Testing .remove. Enter value to remove from list: 5
The list after removing 5 is:
3  10  37  100
Testing client method copyList. The copy list (list2) is:
3  10  37  100
Testing .retrieveAt. First element in list2 is: 3
Testing .replaceAt. Removed first element in list2, added value 25.
After change, list2 is:
10  25  37  100
Was the original list changed? If not, list2 is a deep copy! The original list is:
3  10  37  100
```

```
//Class: ClientOrderedListTime
//Client program to test the OrderedList with Time objects
import java.util.Scanner;
public class ClientOrderedListTime {
    public static final int SIZE = 20;
    public static void main(String[] args)  {
        Scanner input = new Scanner(System.in);
        int h, m, s;
        OrderedArrayList<Time> timeList = new OrderedArrayList<Time>(SIZE);
        OrderedArrayList<Time> temp1;
        OrderedArrayList<Time> temp2;
        timeList.insert(new Time(1, 2, 3));
        timeList.insert(new Time(11, 22, 33));
```

```
        timeList.insert(new Time(1, 12, 13));
        timeList.insert(new Time(4, 5, 6));
        timeList.insert(new Time(11, 12, 13));
        timeList.insert(new Time(7, 8, 9));

        System.out.println("Testing .insert. The original list is: ");
        timeList.print();
        System.out.println("Testing .listSize. The size of the list is: " +
timeList.listSize());
        temp1 = (OrderedArrayList<Time>) timeList.clone();
        System.out.println("Testing .clone. The copy list (temp1) is: ");
        temp1.print();
        temp2 = copyList(timeList);
        System.out.println("Testing client method copyList. The copy list (temp2)
is: ");
        temp2.print();
        Time t = new Time();
        t.setTime(11, 12, 14);
        temp1.insert(t);
        System.out.println("Testing .insert. The copy list (temp1) is: ");
        temp1.print();
        System.out.println("Testing .remove. The copy list (temp1) is: ");
        temp1.remove(t);
        temp1.print();
        t = temp1.retrieveAt(3);
        System.out.println("Testing .remove. The copy list (temp1) is: ");
        temp1.remove(t);
        temp1.print();
        System.out.print("Enter hours minutes and seconds: ");
        h = input.nextInt();
        m = input.nextInt();
        s = input.nextInt();
        t = new Time(h, m, s);
        if (timeList.search(t) != -1)
           System.out.println("Testing .search. The time " + t + " found in original
list.");
        else
           System.out.println("Testing .search. The time " + t + " NOT found in
original list.");
        System.out.println("The original list is: ");
        timeList.print();
        System.out.println("The copy list (temp2) is: ");
        temp2.print();
    }

    public static OrderedArrayList<Time> copyList (OrderedArrayList<Time> otherList)
{
        OrderedArrayList<Time> tempList = new OrderedArrayList<Time>
(otherList.maxListSize());
        Time tl;
        for (int i = 0; i < otherList.listSize(); i++) {
            tl =  otherList.retrieveAt(i);
```

```
            tempList.insert((Time) tl.clone());
        }
        return tempList;
    }
}
```

OUTPUT:

```
Testing .insert. The original list is:
01:02:03  01:12:13  04:05:06  07:08:09  11:12:13  11:22:33
Testing .listSize. The size of the list is: 6
Testing .clone. The copy list (temp1) is:
01:02:03  01:12:13  04:05:06  07:08:09  11:12:13  11:22:33
Testing client method copyList. The copy list (temp2) is:
01:02:03  01:12:13  04:05:06  07:08:09  11:12:13  11:22:33
Testing .insert. The copy list (temp1) is:
01:02:03  01:12:13  04:05:06  07:08:09  11:12:13  11:12:14  11:22:33
Testing .remove. The copy list (temp1) is:
01:02:03  01:12:13  04:05:06  07:08:09  11:12:13  11:22:33
Testing .remove. The copy list (temp1) is:
01:02:03  01:12:13  04:05:06  11:12:13  11:22:33
Enter hours minutes and seconds: 4 5 6
Testing .search. The time 04:05:06 found in original list.
The original list is:
01:02:03  01:12:13  04:05:06  07:08:09  11:12:13  11:22:33
The copy list (temp2) is:
01:02:03  01:12:13  04:05:06  07:08:09  11:12:13  11:22:33
```

---

9. The ArrayList Class

- Although arrays are conceptually important as a data structure, they are not used as much in Java as they are in most other languages. The reason is that the java.util package includes a class called `ArrayList` that provides the standard array behavior along with other useful operations.
- `ArrayList` is a class in the standard Java libraries that can hold any type of object --> In general, an `ArrayList` serves the same purpose as an array, except that an `ArrayList` can change length while the program is running (unlike arrays, which have a fixed length once they have been created).
- The type you specify when creating an `ArrayList` must be an object type; it cannot be a primitive type --> must use wrapper classes!
- Each object of the `ArrayList` class has a capacity. The capacity is the size of the array used to store the elements in the list (it is always at least as large as the list size.) As elements are added to an `ArrayList`, its capacity grows automatically.
- The main difference between Java arrays and `ArrayList` --> `ArrayList` is a Java class rather than a special data type in the language. As a result, all operations on `ArrayList`s are indicated using method calls. For example, the most obvious differences include:

    - You create a new `ArrayList` by calling the `ArrayList` constructor.
    - You get the number of elements by calling the `size` method rather than by selecting a `length` field.
    - You use the `get` and `set` methods to select individual elements.
- Why use an array instead of an `ArrayList`:

    - An `ArrayList` is less efficient than an array

- `ArrayList` does not have the convenient square bracket notation
- The base type of an `ArrayList` must be a class type (or other reference type). It cannot be a primitive type. (Although wrappers, auto-boxing, and auto-unboxing make this a non-issue with most recent versions of Java)

- Why use an `ArrayList` instead of an array?

  - Arrays cant grow. Their size is fixed at compile time, while `ArrayList` grows and shrinks as needed at execution time. The class `ArrayList` is implemented using an array as a private instance variable. When the array is full, a new larger array is created and the data is transferred to this new array (increased capacity)
  - You need to keep track of the actual number of elements in your array, while `ArrayList` will do that for you.
  - Arrays have no methods (just `length` instance variable), while `ArrayList` has powerful methods for manipulating the objects within it.
- **NOTE:** In order to make use of the `ArrayList` class, must add: `import java.util.ArrayList;`
- **NOTE:** The Java standard libraries have a class named `Vector` that behaves almost exactly the same as the class `ArrayList`. In most situations, either class could be used, however the `ArrayList` class is newer (Java 5), and is becoming the preferred class.
- Some **methods** in the `ArrayList` class:

  - `add(value)` --> adds the given value to the end of the list
  - `add(index, value)` --> inserts the given value before the given index
  - `size()` --> returns the number of elements in the list
  - `clear()` --> removes all elements
  - `contains(value)` --> returns true if the given element is in the list
  - `get(index)` --> returns the value at the given index
  - `indexOf(value)` --> returns the first index at which the given element appears in the list (or -1 if not found)
  - `lastIndexOf(value)` --> returns the last index at which the given element appears in the list (or -1 if not found)
  - `remove(index)` --> removes value at given index, sliding others back

Much more in detail (source:http://java.sun.com/javase/6/docs/api/) a more extensive list of methods for the `ArrayList` class (using generics):

`boolean add(E e)` --> Appends the specified element to the end of this list.
`void add(int index, E element)` --> Inserts the specified element at the specified position in this list.
`boolean addAll(Collection<? extends E> c)` --> Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's Iterator.
`boolean addAll(int index, Collection<? extends E> c)` --> Inserts all of the elements in the specified collection into this list, starting at the specified position.
`void clear()` --> Removes all of the elements from this list.
`Object clone()` --> Returns a shallow copy of this ArrayList instance.
`boolean contains(Object o)` --> Returns true if this list contains the specified element.
`void ensureCapacity(int minCapacity)` --> Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
`E get(int index)` --> Returns the element at the specified position in this list.
`int indexOf(Object o)` --> Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element.

`boolean isEmpty()` --> Returns true if this list contains no elements.
`int lastIndexOf(Object o)` --> Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element.
`E remove(int index)` --> Removes the element at the specified position in this list.
`boolean remove(Object o)` --> Removes the first occurrence of the specified element from this list, if it is present.
`protected void removeRange(int fromIndex, int toIndex)` --> Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive.
`E set(int index, E element)` --> Replaces the element at the specified position in this list with the specified element.
`int size()` --> Returns the number of elements in this list.
`Object[] toArray()` --> Returns an array containing all of the elements in this list in proper sequence (from first to last element).
`<T> T[] toArray(T[] a)` --> Returns an array containing all of the elements in this list in proper sequence (from first to last element); the runtime type of the returned array is that of the specified array.
`void trimToSize()` --> Trims the capacity of this ArrayList

- **NOTE:** When looking at the methods available in the `ArrayList` class, there appears to be some inconsistency - in some cases, when a parameter is naturally an object of the base type, the parameter type is the base type; however, in other cases, it is the type `Object`. Why? Because the `ArrayList` class implements a number of interfaces, and inherits methods from various superclasses - they specify that certain parameters have type `Object`.
- **NOTE:** The `ArrayList` class is an example of a collection class (to be discussed later). Starting with version 5.0, Java has added a new kind of **for loop** called a **for-each** or **enhanced for loop.** This kind of loop has been designed to cycle through all the elements in a collection (like an `ArrayList`).
- **Example #1:**

```
import java.util.ArrayList;
public class ExampleForEach implements Cloneable{
    public static void main(String[ ] args){
        ArrayList<Integer> list = new ArrayList<Integer>();
        ArrayList<Integer> copyList = new ArrayList<Integer>();
        list.add(1);
        list.add(2);
        list.add(3);
        System.out.println("The original list:");
        //Reads like: "for each Integer i, in list"
        for(Integer i:list)
            System.out.print(i + " ");
        copyList = (ArrayList)list.clone();
        for(int i = 0; i < copyList.size( ); i++)
            copyList.set(i, new Integer(list.get(i)));
        System.out.println("\nThe clone list:");
        //Reads like: "for each Integer i, in copyList"
        for(Integer i:copyList)
            System.out.print(i + " ");
        System.out.println();
    }
}
```

Output:

```
The original list:
1 2 3
The clone list:
1 2 3
```

- **Example #2:**

```
import java.util.ArrayList;
public class ExampleArrayList implements Cloneable{
    public static void main( String[ ] args) {
        ArrayList<Integer> list = new ArrayList<Integer>(); //initial capacity = 10
        ArrayList<Integer> copyList = new ArrayList<Integer>(); //initial capacity =
10
        System.out.println("Size of original(empty) list = " + list.size());
        for (int i = 1; i <= 10; i++)
            list.add(2 * i);
        System.out.println("Added 10 values to list. The new size is: " +
list.size());
        System.out.println("The list is " + list);
        list.set(0, 59);
        System.out.println("Changed the element with index 0 to 59.");
        System.out.println("The new list is " + list);
        list.add(3, 37);
        System.out.println("Added 37 at index 3. Increased capacity! The new size
is: " + list.size());
        System.out.println("The new list is " + list);
        list.remove(6);
        System.out.println("Removed element with index 6. The new size is: " +
list.size());
        System.out.println("The new list is " + list);
        list.remove((Integer)(6));
        System.out.println("Removed the value 6. The new size is: " + list.size());
        System.out.println("The new list is " + list);
        System.out.println("Printing the list using .get/for loop. The list is: ");
        for (int i = 0; i < list.size(); i++)
            System.out.print(list.get(i) + " " );
        if (list.contains(10))
            System.out.println("\nLooked for value 10 / list. 10 found at index " +
list.indexOf(10));
        else
            System.out.println("Looked for 10/list. 10 NOT found.");
        copyList = (ArrayList)list.clone(); //shallow copy
    //make a deep copy
        for(int i = 0; i < copyList.size( ); i++)
            copyList.set(i, new Integer(list.get(i)));
        System.out.println("\nThe clone list (using for each):");
        for(Integer i:copyList)
            System.out.print(i + " ");
        System.out.println("\nThe clone list (using toString: \n" + copyList);
        System.out.println();
    }
}
```

OUTPUT:

```
Size of original(empty) list = 0
Added 10 values to list. The new size is: 10
The list is [2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Changed the element with index 0 to 59.
The new list is [59, 4, 6, 8, 10, 12, 14, 16, 18, 20]
Added 37 at index 3. Increased capacity! The new size is: 11
The new list is [59, 4, 6, 37, 8, 10, 12, 14, 16, 18, 20]
Removed element with index 6. The new size is: 10
The new list is [59, 4, 6, 37, 8, 10, 14, 16, 18, 20]
Removed the value 6. The new size is: 9
The new list is [59, 4, 37, 8, 10, 14, 16, 18, 20]
Printing the list using .get/for loop. The list is:
59 4 37 8 10 14 16 18 20
Looked for value 10 / list. 10 found at index 4

The clone list (using for each):
59 4 37 8 10 14 16 18 20
The clone list (using toString:
[59, 4, 37, 8, 10, 14, 16, 18, 20]
```

Key Terms

ADT: A data type that specifies the logical properties without the implementation details (information hiding)

Clone method: A method that provides a bit-by-bit copy of the objects data in storage (shallow copy of objects data).

Cloneable: An interface that forces a class to provide an appropriate definition of the method `clone`.

Comparable: An interface used to force a class to provide an appropriate definition of the method `compareTo` so that the values of two objects of that class can be properly compared.

Generic classes (parametric classes): Classes that are used to write a single definition for a set of related classes. Defined using type parameter.

Generic method: Method defined using type parameter.

List: A linear collection of elements of the same type.

(List) Length: The number of elements in the list.

Ordered list: An ordered collection of elements of the same type. A list in which data items are placed in ascending (or descending) order.

Set: A list with distinct elements.

Type parameters (type variables): Identifiers that specify generic type names.

Unordered list: An unordered collection of elements of the same type. A list in which data items are placed in no particular order.

Wrapper class: A class that allows a primitive data type to be instantiated as an object.

Additional Resources:

1. (Sun) API specification for version 6 of the Java Platform, Standard Edition (`Cloneable, Comparable, ArrayList`): http://java.sun.com/javase/6/docs/api/
2. (Sun) The Java Tutorials, Generics in the Java Programming Language (Gilad Bracha): http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf
3. (Sun) The Java Tutorials, Generics: http://java.sun.com/docs/books/tutorial/java/generics/index.html
4. Java Generics FAQs: http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html

---

References:

[1] Java Programming: From Problem Analysis to Program Design, by D.S. Malik, Thomson Course Technology, 2008
[2] Building Java Programs: A Back to Basics Approach, by Stuart Reges, and Marty Stepp, Addison Wesley, 2008.