Basics of the Object-Oriented Programming
Collections

Viorica Rozina Chifu
viorica.chifu@cs.utcluj.ro

Drawing areas - **Canvas** class

- Generic class from AWT used to create drawing surfaces
- Can be extended for implementing graphical objects with a specific representation
- Drawing on a board made by overriding the **paint** method
  - The boards can not contain other graphical components
  - The broads are used as drawing surfaces or as a background for animation
- Board
  - It is a white rectangular area on which we can draw
  - It has default size 0
  - Its dimension can be modified by overriding the methods
    - »**getPreferredSize**, **getMinimumSize**, **getMaximumSize**

Drawing areas - **Canvas** class

- Steps to create a drawing:
  - Create a subclass of **Canvas** class
  - Redefine **paint** method from **Canvas**
  - Overrides **getPreferredSize, getMinimumSize, getMaximumSize** methods from **Canvas**
  - Add the board to a container with **add** method
  - Treat the events: **FocusEvent, KeyEvent, MouseEvent, ComponentEvent**

Drawing areas - **Canvas** class

- Exemple of defining a generic board

```
class Plansa extends Canvas implements...Listener{
   //Constructor
   public Plansa()  {… }
  //Metode de desenare a componentei
   public void paint(Graphics g){… }
  //Metodele folosite de gestionarii de pozitionare
public Dimension getPreferredSize(){
    //Dimensiunea implicita a plansei
      return ...; }
public Dimension getMinimumSize(){ return … }
public Dimension getMaximumSize(){ return ... }
  //Implementarea metodelor interfetelor de tip Listener ... }
```

Drawing areas - **Canvas** class

- Example of drawing

```
class Plansa extends Canvas {
  Dimension dim = new Dimension(100, 100);
  private Color color [] = {Color.red, Color.blue};
  private int index = 0;
  public Plansa (){
     this.addMouseListener(new
MouseAdapter(){
        public void mouseClicked(MouseEvent e){
          index = 1-index;
          repaint(); }});}
 public void paint(Graphics g){
    g.setColor(color[index]);
    g.drawRect(0, 0, dim.width, dim.height);
    g.setColor(color[1-index]);
    g.fillOval(0, 0, dim.width, dim.height); }
 public Dimension getPreferredSize()
   {return dim;}}
```

```
class Fereastra extends JFrame{
    public Fereastra (String titlu){
       super(titlu);
       setSize(200, 200) ;
       add(new Plansa(),
                    BorderLayout.CENTER);
    }
}
 public class TestCanvas {
   public static void main(String args []) {
      new Fereastra ("Test Canvas").
                              setVisible(true);}
}
```

Graphical context for drawing

- Is an object for controlling the drawing process of an object
- Is specified with a **Graphics** object
  - Is transmitted as parameter for **paint** and **update** methods
- **Graphics** class
  - Has methods for:
    - »Drawing graphical primitives
      - Drawing of geometric figures, texts and images
    - »Setting the properties of graphical context
      - Color and font used for drawing
      - The origins of the coordinates of the drawing area
      - Area in which are visible the drawing components
      - How to draw

Graphical context for drawing

- Properties for graphical context
  - Color to draw
    - »Is specified with the methods: **Color getColor**(), void **setColor**(**Color** c)
  - Font to write the text
    - »Is specified with the methods: **Font getFont**(), void **setFont(Font** f)
  - Translating the origin of the coordinate system to the point (x, y) of the current system
    - »Is specified with the method: **translate**(**int** x, **int** y)

Graphical Primitives

- Are drawing with methods from **Graphics** class
  - Allows to draw geometrical figures and texts
- **drawString** method
  - Draws text
    - »The text will be draw with the current font and color of the graphical context
  - Takes as arguments a string and the left-down corner of the text

```
//we draw at the coordinates x=10, y=20;
 drawString("Hello", 10, 20);
```

Graphical Primitives

- Methods for drawing geometrical figures
  - **drawLine**, **drawPolyline** – drawing lines
  - **drawRect**, **fillRect** – drawing simple rectangle/ filled with the current color
  - **draw3DRect**, **fill3DRect** – drawing rectangle with border/ filled with the current color
  - **drawRoundRect, fillRoundRect** – drawing rectangle with rounded corners
  - **drawPolygon, fillPolygon** – polygon drawing
  - **drawOval, fillOval** – ellipse drawing
  - **drawArc, fillArc** – drawing circular or elliptical arc

Graphics2D Class

- This class extends the **Graphics** class to provide more sophisticated control over geometry, coordinate transformations, color management, and text layout
- The main steps in order to draw shapes using Graphics2D class:
  - Create a new Frame
  - Create a class that extends the Component class and override the paint method
  - Use Graphics2D.drawLine to draw a simple line
  - Use Graphics2D.drawOval to draw an oval shape in the screen
  - Use Graphics2D.drawRect to draw a rectangle on the screen

Graphics2D Class -Example

```
import java.awt.*;
public class DrawShapesExample {
  public static void main(String[] args) {
   Frame frame = new Frame();
   // Add a component with a custom paint
method
   frame.add(new
                  CustomPaintComponent());
  //Display the frame
  int frameWidth = 300;
  int frameHeight = 300;
  frame.setSize(frameWidth,
                                frameHeight);
  frame.setVisible(true);
  }
}
```

```
class CustomPaintComponent extends Component
{
  public void paint(Graphics g) {
   //Retrieve the graphics context;
   // this object is used to paint shapes
   Graphics2D g2d = (Graphics2D)g;
   // Draw an oval that fills the window
    int x, y = 0;
   int w = 20;
   int h = 30;
    /** The coordinate system of a graphics context
     is such that the origin is at the northwest corner
     and x-axis increases toward the right while the
     y-axis increases toward the bottom.   */
   g2d.drawLine(x, y, w, h);
   // draw a filled oval
   g2d.drawOval(x, y, w+20, h+30);
   // draw a filled rectangle
  g2d.drawRect(x, y, w+40, h+60);
   }}
```

Drawing - Using fonts

- Writing a text on a screen can be made by:
  - Using a text-oriented component
    - e.g., **Label**
  - Calling the methods for drawing text from **Graphics** class
    - e.g., **drawString**
- We can specify how the text looks through fonts
  - **setFont** method from **Component**, or **Graphics** classes

Drawing - Using fonts

- The parameters that characterize a font:
  - Name of the font: **Helvetica Bold, Arial Bold Italic**, etc.
  - Family to which the font belongs: **Helvetica, Arial**, etc.
  - Dimension of font: Its height
  - Style of the font: **Bold, Italic**
- Classes for working with fonts are:
  - **Font**
  - **FontMetrics**

Drawing - **Font** class

- Constructor of **Font** class

Font(String name, *int* style, *int* size)

  - name – is the name of the font
  - style – is the font syle specified by the following constants:
    - **Font.PLAIN**
    - **Font.BOLD**
    - **Font.ITALIC**
  - size – specifies the font dimension by an integer

new Font("Dialog", Font.PLAIN, 12);
new Font("Arial", Font.ITALIC, 14);
new Font("Courier", Font.BOLD, 10);

Drawing - **Font** class

- Example of using:

```
// For labeled components
JLabel label = new JLabel("Un text");
label.setFont(new Font("Arial", Font.PLAIN,12));
// In paint(Graphics g) method
g.setFont(new Font("Courier", Font.BOLD,10));
g.drawString("Alt text",10, 20);
```

- The list of fonts available on the platform is achieved with the method:

Font[] *fonturi* = **GraphicsEnvironment.getLocalGraphicsEnvironment().getAllFonts**();

Drawing-Using colors

- The color is formed by combining the following colors:
  - Red, green and blue
  - Plus, a certain degree of transparency(**alpha**)
- The color is represented as an instance of **Color** or **SystemColor**
- Each of the four-color parameters varies within a range:
  - »Between 0 and 255 (if we specify the values as integer values)
  - »Between 0.0 and 1.0 (if we specify the values as real values)

Drawing-Using colors

- To create a color, we can use:
  - One of the constants defined in **Color** and **SystemColor** classes
  - One of the constructor of **Color** class
- Constants defined in **Color** class
  - **black, blue, cyan, darkGray, gray, green, lightGray**
- **Color** class defines common colors from standard color palette

  Color red = **Color.red;**

  Color yellow **= Color.yellow;**
- **SystemColor** class
  - Defines the colors of standard components (windows, texts, menus, etc.) of the current work platform

Drawing-Using colors

- Constructors of ***Color*** class

  Color(float red, float green, float blue)

  Color(float red, float green, float blue, float alpha)

  Color(int red, int green, int blue)

  Color(int red, int green, int blue, int alpha)

  - **red, green, blue –** specifies the values for red, green and blue
  - **alpha** - specifies the values for transparency
    - »255 (or 1.0) specifies full opacity of color
    - »0 (sau 0.0) specifies full transparency of color

- Example of using constructors of **Color** class:

```
Color alb = new Color(255, 255, 255);
Color negru = new Color(0, 0, 0);
Color rosu = new Color(255, 0, 0);
Color rosuTransparent = new Color(255, 0, 0, 128);
```

Drawing-Using colors

- Methods defined in **Color** class
  - **Color brighter**(), **Color darker**()
    - »Create a lighter/darker version of current color
  - int **getRed**(), int **getGreen**(), int **getBlue**(), int **getAlpha**()
    - »Determine the parameters used to create the color

Drawing-Using images

- An image is an instance of **Image** class
- The display of an image consists of:
  - Loads an **Image** object with **getImage** method
  - Displays a graphical context whith **drawImage** method from **Graphics** class
    - »Displaying will be made in the **paint** method of a component

Applet	**Toolkit**
getImage(URL url) **getImage(URL** url, String fisier)	**getImage(URL** url) **getImage(String** fisier)

Drawing in Swing

- Is based on AWT model
- **paint** method
  - Is the most important method for drawing
  - Is automatically called every time when is necesarry
  - Is responsible for realizing optimization related to drawing process
  - Has a specific implementation and it mustn't override
- If we want to explicitly redraw a component, we call the method **repaint**
- If the dimension/ position of the component is changed, we call the method **revalidate** followed by **repaint** method

Drawing in Swing

- **paint** method is responsible for calling methods for drawing the components:
  - **paintComponent**
    - »Method for drawing
    - »Is overrrided by each Swing component
  - **paintBorder**
    - »Draw the border of the component
    - »It doesn’t override
  - **paintChildren**
    - »Is used to draw the components contains by this component (if they exist)
    - »It doesn’t override

Java Collections

Collections

- Collection (or container)
  - Object that contains other objects
    - »Collection elements
  - Collection elements can be added/ removed/manipulated in the collection
- The Java Collection framework
  - Represents a unified architecture for storing and manipulating a group of objects
  - Has as elements objects (not primitive types)
  - It has:
    - »Interfaces and its implementations, i.e., classes
    - »Algorithms

Collections

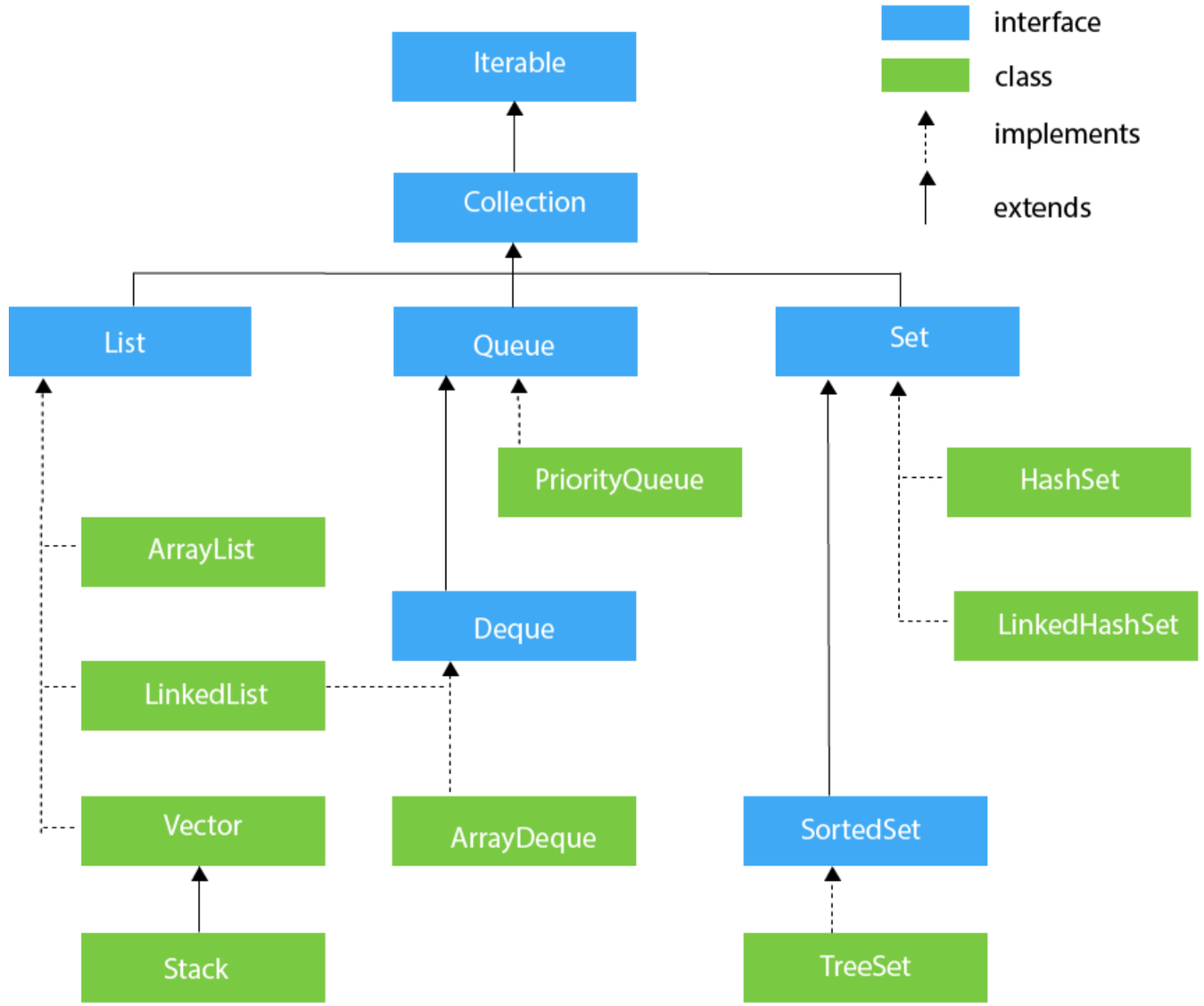
- Java Collection:
  - Any class that stores objects and implements the **Collection** interface
  - For example, the **ArrayList** is collection class, part of the Java collection framework, that implements all methods from **Collection** interface
- Collections are used with iterators
- The **Collection** interface is the highest abstract level in the Java collection framework
- All collection classes are in the java.util package

Collections

- API Collection include:
  - Collections such as **Vector**, **LinkedList**, and **Stack**
  - Maps that index the values based on the keys (e.g., **HashMap)**
  - Variants that ensure that items are always ordered by a comparator: **TreeSet** and **TreeMap**
  - **Iterators** that abstracts the ability to read and write the collections content in loops, and which isolates that ability from implementing the underlying collection

Collections - Collection Interface

- Is the interface which is implemented by all the classes in the collection framework
- It declares the methods that every collection will have



Collections

- Types of collections:
  - Ordered collections
    - »Classes that implement **List** interfaces
  - Collections that ensure element uniqueness
    - »Classes that implement **Set interface**
  - Sorted collection
    - »Classes that implement **SortedSet interface**
  - Collections maintaining the first-in-first-out order
    - »Classes that implements **Queue** interface

Collections - Collection interface

```
interface Collection { // partial list of methods
        public int size();
        public void clear();
        public Object[] toArray();
        public boolean add( Object );
        public boolean remove( Object );
        public boolean removeAll(Collection)
        public boolean addAll( Collection );
        public Iterator iterator();
……................}
```

Collection interface

- Types of operations on Collections
  - Basic Operations
  - Bulk Operations
    - Are operations performed on collections
  - Array Operations
  - Iterator
    - returns an **Iterator** over collection

Collection interface **-** Basic Operations

- **size**
  - Returns the total number of elements in the collection
- **isEmpty**
  - return **true** if the collection is empty (i.e. it doesn't contain elements), otherwise **false**
- **add**
  - Add an element to the end of the collection
- **remove**
  - Removes the specified element from the Collection if it is present and returns a **Boolean** indicating whether the element was present

Collection interface **-** Bulk Operations

- boolean containsAll(Collection c)
  - Returns true if this collection contains all of the elements in the specified collection
- boolean addAll(Collection c)
  - Adds all of the elements in the specified collection to this collection
- boolean removeAll(Collection c)
  - Removes all this collection's elements that are also contained in the specified collection

Collection interface **-** Bulk Operations

- boolean retainAll(Collection c)
  - Retains only the elements in this collection that are contained in the specified collection
- void clear()
  - Removes all of the elements from this collection

Collection interface **-** Array Operations

- Object[] toArray()
  - Returns an array containing all of the elements in this collection
- Object[] toArray(Object[] a)
  - Returns an array containing all of the elements in this collection whose runtime type is that of the specified array

Collection interface **-** Iterator

- Iterator iterator()
  - Returns an iterator over the elements in this collection
- **public interface Iterator**
  - An iterator over a collection
    - »Iterators allow the caller to remove elements from the underlying collection during the iteration with well-defined semantics
  - Contains the following methods
    - »boolean hasNext() - returns true if the iteration has more elements
    - »Object next() - returns the next element in the iteration
    - »void remove() - removes from the underlying collection the last element returned by the iterator

Set interface

- Is present in java.util package
- Extends **Collection** interface
- It models the mathematical set abstraction
  - Is a **Collection** that cannot contain duplicate elements
- It represents the unordered set of elements
- The **Set interface** contains only methods inherited from **Collection** and adds the restriction that duplicate elements are prohibited

Set interface

- Set is implemented by HashSet, LinkedHashSet, and TreeSet classes
  - **HashSet**
    - »It inherits the **AbstractSet** class and implements **Set** interface
    - »It represents the collection that uses a hash table for storage
    - »Hashing is used to store the elements in the **HashSet**
    - »It contains unique items
    - »HashSet allows null value
    - »HashSet class is non synchronized
    - »HashSet doesn't maintain the insertion order
      - Elements are inserted on the basis of their hashcode
    - »HashSet is the best approach for search operations

Set interface

- Set is implemented by HashSet, LinkedHashSet, and TreeSet classes
  - **LinkedHashSet**
    - »It extends **HashSet** class which implements **Set** interface
    - »Is the **LinkedList** implementation of **Set** Interface
    - »Like **HashSet**, it also contains unique element
    - »It maintains the insertion order and permits null elements
    - »It is non synchronized

Set interface

- Set is implemented by HashSet, LinkedHashSet, and TreeSet classes
  - **TreeSet**
    - »It implements the **Set** interface that uses a tree for storage
    - »It implements the **NavigableSet** interface that extends **SortedSet**, **Set**, **Collection** and **Iterable** interfaces in hierarchical order
    - »Like **HashSet**, **TreeSet** also contains unique elements
    - »The access and retrieval time of **TreeSet** is quite fast
    - »The elements in **TreeSet** stored in ascending order
    - »It doesn't allow null element
    - »It is non synchronized

Set interface

- Set can be instantiated as:

```
Set<data-type> s1 = new HashSet<data-type>();
Set<data-type> s2 = new LinkedHashSet<data-type>();
Set<data-type> s3 = new TreeSet<data-type>();
```

Set interface **–** Basic Operations (**Example)**

```
import java.util.*;
public class TestCollection{
 public static void main(String args[]){
        //Creating HashSet and adding elements
        HashSet<String> set=new HashSet<String>();
        set.add(“Ana");
        set.add("Victor");
        set.add(“Ana");
        set.add(“Dan");
        //Traversing elements
        Iterator<String> itr=set.iterator();
        while(itr.hasNext()){
                System.out.println(itr.next());
        System.out.println(“ number of distinct names”+ set.size());
} }
}
```

Set interface – Bulk Operations

- To calculate the **union**, **intersection**, or **difference** of two sets without modifying either set, a copy of the set before calling the appropriate bulk operation need to be kept
- The following examples alters the union, intersection and difference sets:

```
Set<Type> union= new HashSet<Type>s1
union.addAll(s2);

Set<Type> intersection= new HashSet<Type>s1
 intersection.addAll(s2);

Set<Type> difference= new HashSet<Type>s1
difference.addAll(s2);
```

Set interface – Examples

- Consider the following example:

```
import java.util.*;
 public class FindDups {
    public static void main(String[] args) {
        Set<String> s = new HashSet<String>();
        for (String a : args)
                s.add(a);
                System.out.println(s.size() + " distinct words: " + s);
    }
}
```

Now run either version of the program.

```
java FindDups i came i saw i left
```

The following output is produced:

```
4 distinct words: [left, came, saw, i]
```

Set interface – Examples

- For the considered example
  - Suppose you want to know which
    - » Words in the argument list occur only once
    - » Occur more than once, but you do not want any duplicates printed out repeatedly
  - This can be achieved by generating two sets
    - » One containing every word in the argument list and
    - » The other containing only the duplicates
  - The words that occur only once are computed as the difference between the set containing all the words and the set containing duplicates

Set interface **-** Examples

- Revisit **FindDups** example

```
import java.util.*;
 public class FindDups {
    public static void main(String[] args) {
        Set<String> unique = new HashSet<String>();
        Set<String> dups = new HashSet<String>();
        for (String a : args)
              if(! unique.add(a))
                 dup.add(a);
        unique.removeAll(dup)
        System.out.println(" unique words: " + unique);
        System.out.println(" dup words: " + dup);
  }
}
```

  - When run with the same argument list used earlier (i came i saw i left), the program yields the following output:

```
Unique words:    [left, saw, came]
Duplicate words: [i]
```

SortedSet Interface

- A **SortedSet** is a **Set** that maintains its elements in ascending order, sorted
  - According to the elements' natural ordering
  - According to a **Comparator** provided at **SortedSet** creation time
- In addition to the normal **Set** operations, the **SortedSet** interface provides operations for:
  - Range view — allows arbitrary range operations on the sorted set
  - Endpoints — returns the first or last element in the sorted set
  - Comparator access — returns the Comparator used to sort the set

SortedSet Interface

- SortedSet interface declares the following methods:

```
public interface SortedSet<E> extends Set<E> {
    // Range-view
    SortedSet<E> subSet(E fromElement, E toElement);
    SortedSet<E> headSet(E toElement);
    SortedSet<E> tailSet(E fromElement);
    // Endpoints
    E first();
    E last();

    // Comparator access
    Comparator<? super E> comparator();
}
```

SortedSet Interface

- SortedSet<E> subSet(E fromElement, E toElement);
  - Returns a sorted subset from the set containing the elements between element1 and element2.
- SortedSet<E> headSet(E toElement)
  - Returns the elements which are less than toElement
- SortedSet<E> tailSet(E fromElement)
  - Returns the elements which are greater than or equal to fromElement

SortedSet Interface

- E first()
  - Returns the first element of the set
- E last();
  - Returns the last element of the set
- Comparator<? super E> comparator();
  - Returns the comparator used to order the elements in this set, or null if this set uses the natural ordering of its elements

List interface

- Is the child interface of Collection interface
- It inhibits a list type data structure in which we can store the ordered collection of objects
- It can have duplicate values
- List interface is implemented by the classes ArrayList, LinkedList, Vector, and Stack

List interface

- To instantiate the List interface, we must use :

  List <data-type> list1= **new** ArrayList();

  List <data-type> list2 = **new** LinkedList();

  List <data-type> list3 = **new** Vector();

  List <data-type> list4 = **new** Stack();

List interface

- In addition to the operations inherited from **Collection**, the **List** interface includes operations for:
  - Positional access
    - » Manipulates elements based on their numerical position in the list
      - This includes methods such as **get**, **set**, **add**, **addAll**, and **remove**
  - Search
    - »Searches for a specified object in the list and returns its numerical position
      - Search methods include **indexOf** and **lastIndexOf**
  - Iteration
    - »Extends **Iterator** semantics to take advantage of the list's sequential nature
      - The **ListIterator** methods provide this behavior
  - Range-view
    - »The **sublist** method performs arbitrary range operations on the list

List - Positional Access and Search Operations

- E **get**(int index)
  - Returns the element at the specified position in this list
- E **set(int** index, E element)
  - Replaces the element at the specified position in this list with the specified element
  - Return the old value that is being overwritten
- void **add(int** index, E element)
  - Inserts the specified element at the specified position in the list
- E remove(int index)
  - Removes the element at the specified position in this list
  - Return the old value that is removed

List - Positional Access and Search Operations

- 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
- 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
- **addAll(int** index, Collection<? extends E> c)
  - Inserts all the elements of the specified Collection starting at the specified position
  - The elements are inserted in the order they are returned by the specified Collection's iterator

List - Positional Access and Search Operations (Example)

- Example of a method to swap two indexed values in a **List**

```
public static void swap(List<E> a, int i, int j) {
   E tmp = a.get(i);
   a.set(i, a.get(j));
   a.set(j, tmp); }
```

- Example of polymorphic algorithm that uses the preceding swap method

```
public static void shuffle(List<?> list, Random rnd) {
    for (int i = list.size(); i > 1; i--)
        swap(list, i - 1, rnd.nextInt(i)); }
```

List - Positional Access and Search Operations (Example)

- Example that using shuffle method from collection to print the words in its argument list in random order

```
import java.util.*;
public class Shuffle {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>();
        for (String a : args)
            list.add(a);
        Collections.shuffle(list, new Random());
        System.out.println(list);
    }}
```

List - Positional Access and Search Operations (Example)

- The example from the previous slide can be made shorter and faster
  - By using **asList** static method from **Arrays** class which allows an array to be viewed as a **List**
  - The resulting **List** is not a general-purpose **List** implementation, because it doesn't implement the (optional) **add** and **remove** operations: Arrays are not resizable

```
import java.util.*;
public class Shuffle {
    public static void main(String[] args) {
        List<String> list = Arrays.asList(args);
        Collections.shuffle(list);
        System.out.println(list);
    }}
```

List - Range-View Operation

- **subList**(**int** fromIndex, **int** toIndex)
  - Returns a **List** view of the portion of this list whose indices range from *fromIndex*, inclusive, to *toIndex*, exclusive
- Any operation that expects a **List** as argument can be called on the result provided by **subList** method
  - For example, the following example removes a range of elements from a **List**

```
list.subList(fromIndex, toIndex).clear();
```

  - Similar examples can be constructed to search for an element in a range

```
int i = list.subList(fromIndex, toIndex).indexOf(o);
int j = list.subList(fromIndex, toIndex).lastIndexOf(o);
```

List - Algorithms

- Most polymorphic algorithms in the **Collections** class apply specifically to List
- Example of methods defined in Collections class:
  - **sort**
    - »sorts a **List** using a merge sort algorithm, which provides a fast, stable sort
  - **shuffle**
    - » randomly permutes the elements in a **List**
  - **revers**
    - » reverses the order of the elements in a **List**
  - **rotate**
    - » rotates all the elements in a List by a specified distance
  - **swap**
    - »swaps the elements at specified positions in a List

List - Algorithms

- Example of methods defined in **Collections** class:
  - **replaceAll**
    - » replaces all occurrences of one specified value with another
  - **fill**
    - »overwrites every element in a **List** with the specified value
  - **copy**
    - » copies the source **List** into the destination **List**
  - **binarySearch**
    - »searches for an element in an ordered **List** using the **binary** search algorithm
  - **indexOfSubList**
    - » returns the index of the first sublist of one **List** that is equal to another
  - **lastIndexOfSubList**
    - »returns the index of the last sublist of one List that is equal to another