Appendix 4
Summary of Classes and Interfaces

This appendix summarizes most of the library classes used in this book. This appendix includes some methods, and even some classes, that are not discussed in the text. However, the lists of class methods and other class members are the most commonly used members and the members used in this book, but they are not complete lists of methods for the classes given here.

If a class or interface is derived from another class or interface, respectively, then in some cases, the table for the derived class or interface lists only new methods and does not list all the inherited methods.

ABSTRACT BUTTON

Package: `javax.swing`

The classes `JButton` and `JMenuItem` are also in this package.

All these methods are inherited by the classes `JButton` and `JMenuItem`.

`AbstractButton` is an abstract class.

Ancestor classes:

```
Object
   |
   +--Component
         |
         +--Container
               |
               +--JComponent
                     |
                     +--AbstractButton
```

Implements Interfaces: `ImageObserver, ItemSelectable, MenuContainer, Serializable, SwingConstants`

```
public void addActionListener(ActionListener listener)
```

Adds an `ActionListener`.

```
public String getActionCommand()
```

Returns the action command for this component.

```
public String getText()
```

Returns the text written on the component, such as the text on a button or the string for a menu item.

```
public void removeActionListener(ActionListener listener)
```

Removes an `ActionListener`.

```
public void setActionCommand(String actionCommand)
```

Sets the action command.

```
public void setBackground(Color theColor)
```

Sets the background color of this component.

```
public void setMaximumSize(Dimension maximumSize)
```

Sets the maximum size of the button or label. Note that this is only a suggestion to the layout manager. The layout manager is not required to respect this maximum size. The following special case will work for most simple situations. The `int` values give the width and height in pixels.

```
public void setMaximumSize(
                        new Dimension(int width, int height))
```

```
public void setMinimumSize(Dimension minimumSize)
```

Sets the minimum size of the button or label. Note that this is only a suggestion to the layout manager. The layout manager is not required to respect this minimum size.

Although we do not discuss the `Dimension` class, the following special case is intuitively clear and will work for most simple situations. The `int` values give the width and height in pixels.

```
public void setMinimumSize(
                         new Dimension(int width, int height))
```

```
public void setPreferredSize(Dimension preferredSize)
```

Sets the preferred size of the button or label. Note that this is only a suggestion to the layout manager. The layout manager is not required to use the preferred size. The following special case will work for most simple situations. The `int` values give the width and height in pixels.

```
public void setPreferredSize(
                        new Dimension(int width, int height))
```

```
public void setText(String text)
```

Makes text the only text on this component.

Boolean

See Section 5.1 in Chapter 5.

BufferedReader

Package: `java.io`

The `FileReader` class is also in this package.

Ancestor classes:

```
Object
   |
   +--Reader
         |
         +--BufferedReader
```

```
public BufferedReader(Reader readerObject)
```

This is the only constructor you are likely to need. There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you use

```
new BufferedReader(new FileReader(File_Name))
```

When used in this way, the `FileReader` constructor, and thus the `BufferedReader` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

If you want to create a stream using an object of the class `File`, you use

```
new BufferedReader(new FileReader(File_Object))
```

When used in this way, the `FileReader` constructor, and thus the `BufferedReader` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

```
public void close() throws IOException
```

Closes the stream's connection to a file.

```
public int read() throws IOException
```

Reads a single character from the input stream and returns that character as an `int` value. If the read goes beyond the end of the file, then –1 is returned. Note that the value is returned as an `int`. To obtain a `char`, you must perform a type cast on the value returned. The end of a file is signaled by returning –1. (All of the "real" characters return a positive integer.)

```
public String readLine() throws IOException
```

Reads a line of input from the input stream and returns that line. If the read goes beyond the end of the file, `null` is returned. (Note that an `EOFException` is not thrown at the end of a file. The end of a file is signaled by returning `null`.)

```
public long skip(long n) throws IOException
```

Skips n characters.

Byte

See Section 5.1 in Chapter 5.

Character

Package: `java.lang`

Ancestor classes:

```
Object
   |
   +--Character
```

Implemented Interfaces: `Comparable`, `Serializable`

The `Character` class is marked `final`, which means it cannot be used as a base class to derive other classes.

```
public static boolean isDigit(char argument)
```

Returns `true` if its `argument` is a digit; otherwise returns `false`.

EXAMPLES

`Character.isDigit('5')` returns `true`. `Character.isDigit('A')` and `Character.isDigit('%')` both return `false`.

```
public static boolean isLetter(char argument)
```

Returns `true` if its `argument` is a letter; otherwise returns `false`.

EXAMPLES

`Character.isLetter('A')` returns `true`. `Character.isLetter('%')` and `Character.isLetter('5')` both return `false`.

```
public static boolean isLetterOrDigit(char argument)
```

Returns `true` if its `argument` is a letter or a digit; otherwise returns `false`.

EXAMPLES

`Character.isLetterOrDigit('A')` and `Character.isLetterOrDigit('5')` both return `true`. `Character.isLetterOrDigit('&')` returns `false`.

```
public static boolean isLowerCase(char argument)
```

Returns `true` if its `argument` is a lowercase letter; otherwise returns `false`.

EXAMPLES

`Character.isLowerCase('a')` returns `true`. `Character.isLowerCase('A')` and `Character.isLowerCase('%')` both return `false`.

```
public static boolean isUpperCase(char argument)
```

Returns `true` if its `argument` is an uppercase letter; otherwise returns `false`.

EXAMPLES

`Character.isUpperCase('A')` returns `true`. `Character.isUpperCase('a')` and `Character.isUpperCase('%')` both return `false`.

`public static boolean isWhitespace(char argument)`

Returns `true` if its `argument` is a whitespace character; otherwise returns `false`. Whitespace characters are those that print as white space, such as the space character (blank character), the tab character (`'\t'`), and the new-line character (`'\n'`).

EXAMPLES

`Character.isWhitespace(' ')` returns `true`. `Character.isWhitespace('A')` returns `false`.

`public static char toLowerCase(char argument)`

Returns the lowercase version of its `argument`. If the `argument` is not a letter, it is returned unchanged.

EXAMPLE:

`Character.toLowerCase('a')` and `Character.toLowerCase('A')` both return `'a'`.

`public static char toUpperCase(char argument)`

Returns the uppercase version of its `argument`. If the `argument` is not a letter, it is returned unchanged.

EXAMPLE:

`Character.toUpperCase('a')` and `Character.toUpperCase('A')` both return `'A'`.

`Collection` INTERFACE

Package: `java.util`

Ancestor interfaces: None

All the exception classes mentioned are unchecked exceptions, which means they are not required to be caught in a `catch` block or declared in a `throws` clause.

All the exception classes mentioned are in the package `java.lang` and so do not require any `import` statement.

CONSTRUCTORS

Although not officially required by the interface, any class that implements the `Collection` interface should have at least two constructors: a no-argument constructor that creates an empty `Collection` object, and a constructor with one parameter of type `Collection` that creates a `Collection` object with the same elements as the constructor argument. The interface does not specify whether the copy produced by the one-argument constructor is a shallow copy or a deep copy of its argument.

`public boolean contains(Object target)`

Returns `true` if the calling object contains at least one instance of `target`. Uses `target.equals` to determine if `target` is in the calling object.

Throws:

`ClassCastException` if the type of `target` is incompatible with the calling object (optional).

`NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

```
public boolean containsAll(Collection collectionOfTargets)
```

Returns `true` if the calling object contains all of the elements in `collectionOfTargets`. For `element` in `collectionOfTargets`, this method uses `element.equals` to determine if `element` is in the calling object.

Throws:

`ClassCastException` if the types of one or more elements in `collectionOfTargets` are incompatible with the calling object (optional).

`NullPointerException` if `collectionOfTargets` contains one or more `null` elements and the calling object does not support `null` elements (optional).

`NullPointerException` if `collectionOfTargets` is `null`.

```
public boolean equals(Object other)
```

This is the `equals` of the collection, not the `equals` of the elements in the collection. Overrides the inherited method `equals`. Although there are no official constraints on `equals` for a collection, it should be defined as we have described in Chapter 7 and also to satisfy the intuitive notion of collections being equal.

```
public int hashCode()
```

Returns the hash code value for the calling object. Neither hash codes nor this method is discussed in this book. This entry is only here to make the definition of the `Collection` interface complete. You can safely ignore this entry until you go on to study hash codes in a more advanced book. In the meantime, if you need to implement this method, have the method throw an `UnsupportedOperationException`.

```
boolean isEmpty()
```

Returns `true` if the calling object is empty; otherwise returns `false`.

```
Iterator iterator()
```

Returns an iterator for the calling object. (Iterators are discussed in Section 15.3.)

```
public Object[] toArray()
```

Returns an array containing all of the elements in the calling object. If the calling object makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order.

The array returned should be a new array so that the calling object has no references to the returned array. (You might also want the elements in the array to be clones of the elements in the collection. However, this is apparently not required by the interface, since library classes, such as `Vector`, return arrays that contain references to the elements in the collection.)

```
public Object[] toArray(Object[] a)
```

Returns an array containing all of the elements in the calling object. The argument `a` is used primarily to specify the type of the array returned. The exact details are as follows:

The type of the returned array is that of `a`. If the elements in the calling object fit in the array `a`, then `a` is used to hold the elements of the returned array; otherwise a new array is created with the same type as `a`.

If `a` has more elements than the calling object, the element in `a` immediately following the end of the copied elements is set to `null`.

If the calling object makes any guarantees as to what order its elements are returned by its iterator, this method must return the elements in the same order. (Iterators are discussed in Section 15.3.)

Throws:

`ArrayStoreException` if the type of *a* is not an ancestor type of the type of every element in the calling object.

`NullPointerException` if *a* is `null`.

`public int size()`

Returns the number of elements in the calling object. If the calling object contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

Optional Methods

The following methods are optional, which means they still must be implemented, but the implementation can simply throw an `UnsupportedOperationException` if for some reason you do not want to give them a "real" implementation. An `UnsupportedOperationException` is a `RunTimeException` and so is not required to be caught or declared in a `throws` clause.

`public boolean add(Object element)` *(Optional)*

Ensures that the calling object contains the specified `element`. Returns `true` if the calling object changed as a result of the call. Returns `false` if the calling object does not permit duplicates and already contains `element`.

Throws:

`UnsupportedOperationException` if this method is not supported by the class that implements this interface.

`ClassCastException` if the class of `element` prevents it from being added to the calling object.

`NullPointerException` if `element` is `null` and the calling object does not support `null` elements.

`IllegalArgumentException` if some other aspect of `element` prevents it from being added to the calling object.

`public boolean addAll(Collection collectionToAdd)` *(Optional)*

Ensures that the calling object contains all the elements in `collectionToAdd`. Returns `true` if the calling object changed as a result of the call; returns `false` otherwise.

Throws:

`UnsupportedOperationException` if this method is not supported by the class that implements this interface.

`ClassCastException` if the class of an element of `collectionToAdd` prevents it from being added to the calling object.

`NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.

`IllegalArgumentException` if some aspect of an element of `collectionToAdd` prevents it from being added to the calling object.

`public void clear()` *(Optional)*

Removes all the elements from the calling object.

Throws:

`UnsupportedOperationException` if this method is not supported by the class that implements this interface.

`public boolean remove(Object element)` *(Optional)*

Removes a single instance of the `element` from the calling object, if it is present. Returns `true` if the calling object contained the `element`; returns `false` otherwise.

Throws:

`UnsupportedOperationException` if this method is not supported by the class that implements this interface.

`ClassCastException` if the type of `element` is incompatible with the calling object (optional).

`NullPointerException` if `element` is `null` and the calling object does not support `null` elements (optional).

`public boolean removeAll(Collection collectionToRemove)` *(Optional)*

Removes all the calling object's elements that are also contained in `collectionToRemove`. Returns `true` if the calling object was changed; otherwise returns `false`.

Throws:

`UnsupportedOperationException` if this method is not supported by the class that implements this interface.

`ClassCastException` if the types of one or more elements in `collectionToRemove` are incompatible with the calling collection (optional).

`NullPointerException` if `collectionToRemove` contains one or more `null` elements and the calling object does not support `null` elements (optional).

`NullPointerException` if `collectionToRemove` is `null`.

`public boolean retainAll(Collection saveElements)`

Retains only the elements in the calling object that are also contained in the collection `saveElements`. In other words, removes from the calling object all of its elements that are not contained in the collection `saveElements`. Returns `true` if the calling object was changed; otherwise returns `false`.

Throws:

`ClassCastException` if the types of one or more elements in `saveElements` are incompatible with the calling object (optional).

`NullPointerException` if `saveElements` contains one or more `null` elements and the calling object does not support `null` elements (optional).

`NullPointerException` if `saveElements` is `null`.

Color

Package: `java.awt`

Ancestor classes:

```
Object
   |
   +--Color
```

Implements Interfaces: `Paint, Serializable, Transparency`

Constructors

`public Color(float r, float g, float b)`

Constructor that creates a new `Color` with the specified RGB values. The parameters `r`, `g`, and `b` must each be in the range `0.0` to `1.0` (inclusive).

`public Color(int r, int g, int b)`

Constructor that creates a new `Color` with the specified RGB values. The parameters `r`, `g`, and `b` must each be in the range `0` to `255` (inclusive).

Methods

`public Color brighter()`

Returns a brighter version of the calling object color.

`public Color darker()`

Returns a darker version of the calling object color.

`public boolean equals(Object c)`

Returns `true` if `c` is equal to the calling object color; otherwise returns `false`.

`public int getBlue()`

Returns the blue component of the calling object. The returned value is in the range `0` to `255` (inclusive).

`public int getGreen()`

Returns the green component of the calling object. The returned value is in the range `0` to `255` (inclusive).

`public int getRed()`

Returns the red component of the calling object. The returned value is in the range `0` to `255` (inclusive).

Constants

`Color.BLACK`
`Color.BLUE`
`Color.CYAN`
`Color.DARK_GRAY`
`Color.GRAY`
`Color.GREEN`
`Color.LIGHT_GRAY`
`Color.MAGENTA`
`Color.ORANGE`
`Color.PINK`
`Color.RED`
`Color.WHITE`
`Color.YELLOW`

`Comparable` INTERFACE

package: `java.lang`

The `Comparable` interface has only one method heading that must be implemented.

```
public int compareTo(Object other)
```

The method `compareTo` should return

a negative number if the calling object "comes before" the parameter `other`,

a zero if the calling object "equals" the parameter `other`,

and a positive number if the calling object "comes after" the parameter `other`.

The "comes before" ordering that underlies `compareTo` should be a total ordering. Most normal ordering, such as less-than on numbers and lexicographic ordering on strings, are total orderings

`ConsoleIn`

See Section 2.4 in Chapter 2.

`Double`

See Section 5.1 in Chapter 5.

`File`

Package: `java.io`

Ancestor classes:

```
Object
   |
   +--File
```

Implemented Interfaces: `Comparable, Serializable`

Many of these methods throw a `SecurityException` if a security manager exists and is unhappy with the method invocation. This is not likely to be a concern for readers of this book and we have not noted this in the method descriptions.

The class `SecurityException` is an unchecked exception class, which means you need not catch it or declare it in a `throws` clause.

```
public File(String fileName)
```

Constructor. `fileName` can be either a full or a relative path name (which includes the case of a simple file name). `fileName` is referred to as the **abstract path name**.

Throws:

`NullPointerException` if the pathname `fileName` is `null`.

```
public boolean canRead()
```

Tests whether the program can read from the file. Returns `true` if the file named by the abstract path name exists and is readable by the program; otherwise returns `false`.

`public boolean canWrite()`

Tests whether the program can write to the file. Returns `true` if the file named by the abstract path name exists and is writable by the program; otherwise returns `false`.

`public boolean createNewFile()`

Creates a new empty file named by the abstract path name, provided that a file of that name does not already exist. Returns `true` if successful; returns `false` otherwise.

Throws:

`IOException` if an I/O error occurs.

`public boolean delete()`

Tries to delete the file or directory named by the abstract path name. A directory must be empty to be removed. Returns `true` if it was able to delete the file or directory. Returns `false` if it was unable to delete the file or directory.

`public boolean exists()`

Tests whether there is a file with the abstract path name.

`public String getName()`

Returns the last name in the abstract path name (that is, the simple file name). Returns the empty string if the abstract path name is the empty string.

`public String getPath()`

Returns the abstract path name as a `String` value.

`public boolean isDirectory()`

Returns `true` if a directory (folder) exists that is named by the abstract path name; otherwise returns `false`.

`public boolean isFile()`

Returns `true` if a file exists that is named by the abstract path name and the file is a normal file; otherwise returns `false`. The meaning of *normal* is system dependent. Any file created by a Java program is guaranteed to be normal.

`public long length()`

Returns the length in bytes of the file named by the abstract path name. If the file does not exist or the abstract path name names a directory, then the value returned is not specified and may be anything.

`public boolean mkdir()`

Makes a directory named by the abstract path name. Will not create parent directories. See `mkdirs`. Returns `true` if successful; otherwise returns `false`.

`public boolean mkdirs()`

Makes a directory named by the abstract path name. Will create any necessary but nonexistent parent directories. Returns `true` if successful; otherwise returns `false`. Note that if it fails, then some of the parent directories may have been created.

`public boolean renameTo(File newName)`

Renames the file represented by the abstract path name to `newName`. Returns `true` if successful; otherwise returns `false`. `newName` can be a relative or absolute path name. This may require moving the file. Whether or not the file can be moved is system dependent.

Throws:

`NullPointerException` if parameter `newName` is null.

`public boolean setReadOnly()`

Sets the file represented by the abstract path name to be read only. Returns `true` if successful; otherwise returns `false`.

Float

See Section 5.1 in Chapter 5.

Font

Package: `java.awt`

Implements Interface: `Serializable`

Ancestor classes:

```
Object
   |
   +--Font
```

CONSTRUCTOR

`public Font(String fontName, int styleModifications, int size)`

Constructor that creates a version of the font named by `fontName` with the specified `styleModifications` and `size`.

CONSTANTS

`Font.BOLD`

Specifies bold style.

`Font.ITALIC`

Specifies italic style.

`Font.PLAIN`

Specifies plain style—that is, not bold and not italic.

Names of Fonts

(These three are guaranteed by Java. Your system will probably have others as well as these.)

`"Monospaced"`

See Chapter 18 for a sample.

`"SansSerif"`

See Chapter 18 for a sample.

`"Serif"`

See Chapter 18 for a sample.

Method that uses Font

```
public abstract void setFont(Font fontObject)
```

This method is in the class `Graphics`. Sets the current font of the calling `Graphics` object to `fontObject`.

Graphics

Package: `java.awt`

Ancestor classes:

```
Object
   |
   +--Graphics
```

`Graphics` is an abstract class.

Although many of these methods are abstract, we always use them with objects of a concrete descendant class of `Graphics`, even though we usually do not know the name of that concrete class.

```
public abstract void drawRect(int x, int y,
                              int width, int height)
```

Draws the outline of the specified rectangle. (x, y) is the location of the upper-left corner of the rectangle.

```
public abstract void fillRect(int x, int y,
                              int width, int height)
```

Fills the specified rectangle. (x, y) is the location of the upper-left corner of the rectangle.

```
public void draw3DRect(int x, int y, int width,
                                    int height, boolean raised)
```

Draws the outline of the specified rectangle. (x, y) is the location of the upper-left corner. The rectangle is highlighted to look like it has thickness. If `raised` is `true`, the highlight makes the rectangle appear to stand out from the background. If `raised` is `false`, the highlight makes the rectangle appear to be sunken into the background.

```
public void fill3DRect(int x, int y, int width,
                                    int height, boolean raised)
```

Fills the rectangle specified by

```
draw3DRec(x, y, width, height, raised)
```

```
public abstract void drawArc(int x, int y,
                             int width, int height,
                             int startAngle, int arcSweep)
```

Draws part of an oval that just fits into an invisible rectangle described by the first four arguments. The portion of the oval drawn is given by the last two arguments. See Chapter 18 for details.

```
public abstract void drawLine(int x1, int y1, int x2, int y2)
```

Draws a line between points (x1, y1) and (x2, y2).

```
public abstract void drawOval(int x, int y,
                              int width, int height)
```

Draws the outline of the oval with the smallest enclosing rectangle that has the specified width and height. The (imagined) rectangle has its upper-left corner located at (x, y).

```
public void drawPolygon(int[] x, int[] y, int points)
```

Draws a polygon through the point
`(x[0], y[0]), (x[1], y[1]), ..., (x[points – 1], y[points – 1])`.
Always draws a closed polygon. If the first and last points are not equal, it draws a line from the last to the first point.

```
public void drawPolyline(int[] x, int[] y, int points)
```

Draws a polygon through the point
`(x[0], y[0]), (x[1], y[1]), ..., (x[points – 1], y[points – 1])`.
If the first and last points are not equal, the polygon will not be closed.

```
public abstract void drawRoundRect(int x, int y,
             int width, int height, int arcWidth, int arcHeight)
```

Draws the outline of the specified round-cornered rectangle. (x, y) is the location of the upper-left corner of the enclosing regular rectangle. `arcWidth` and `arcHeight` specify the shape of the round corners. See the text for details.

```
public abstract void drawString(String text, int x, int y)
```

Draws the `text` given by the specified string, using this graphics object's current font and color. The baseline of the leftmost character is at position (x, y) in this graphics object's coordinate system.

```
public abstract void fillArc(int x, int y,
                             int width, int height,
                             int startAngle, int arcSweep)
```

Fills the partial oval specified by

```
drawArc(x, y, width, height, startAngle, arcSweep)
```

```
public abstract void fillOval(int x, int y,
                              int width, int height)
```

Fills the oval specified by

```
drawOval(x, y, width, height)
```

```
public void fillPolygon(int[] x, int[] y, int points)
```

Fills (with color) the polygon specified by `drawPolygon(x,y,points)`.

```
public abstract void fillRoundRect(int x, int y,
            int width, int height, int arcWidth, int arcHeight)
```

Fills the round rectangle specified by

```
drawRoundRec(x, y, width, height, arcWidth, arcHeight)
```

```
public abstract void setFont(Font fontObject)
```

Sets the current font of the calling `Graphics` object to `fontObject`.

HashSet

Package: `java.util`

Ancestor classes:

```
Object
   |
   +--AbstractCollection
         |
         +--AbstractSet

                 |
                 +--HashSet
```

Implements Interfaces: `Cloneable, Collection, Serializable, Set`

The `HashSet` class implements all of the methods in the `Set` interface. The only other methods in the `HashSet` class are the constructors. The two constructors that do not involve concepts beyond the scope of this book are given below.

All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.

All the exception classes mentioned are in the package `java.lang` and so do not require any `import` statement.

`public HashSet()`

Creates a new, empty set.

`public HashSet(Collection c)`

Creates a new set that contains all the elements of `c`.

Throws:

`NullPointerException` if `c` is `null`.

`public HashSet(int initialCapacity)`

Creates a new, empty set with the specified capacity.

Throws:

`IllegalArgumentException` if `initialCapacity` is less than zero.

The methods are the same as those described for the `Set` interface.

Integer

See Section 5.1 in Chapter 5.

Iterator INTERFACE

package: `java.util`

All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.

`NoSuchElementException` is in the `java.util` package, which requires an `import` statement if your code mentions the `NoSuchElementException` class. All the other exception classes mentioned are in the package `java.lang` and so do not require any `import` statements.

`public boolean hasNext()`

Returns `true` if `next()` has not yet returned all the elements in the collection; returns `false` otherwise.

`public Object next()`

Returns the next element of the collection that produced the iterator.

Throws:

`NoSuchElementException` if there is no next element.

`public void remove()` *(Optional)*

Removes from the collection the last element returned by `next`.

This method can be called only once per call to `next`.

Throws:

`IllegalStateException` if the `next` method has not yet been called, or the `remove` method has already been called after the last call to the `next` method.

`UnsupportedOperationException` if the `remove` operation is not supported by this `Iterator`.

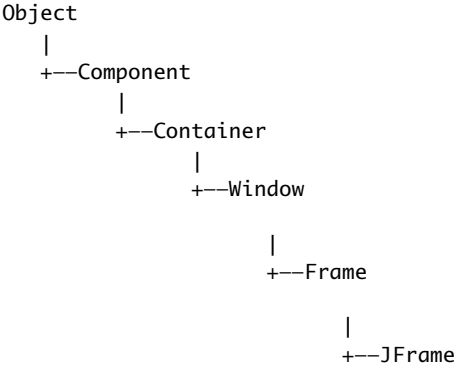
JButton

See AbstractButton.

JFrame

Package: javax.swing

Ancestor classes:



Implements Interfaces: Accessible, ImageObserver, MenuContainer, RootPaneContainer, Serializable, WindowConstants

```
public JFrame()
```

Constructor that creates an object of the class JFrame.

```
public JFrame(String title)
```

Constructor that creates an object of the class JFrame with the title given as the argument.

```
public Container getContentPane()
```

Returns the content pane of the calling JFrame object. Container is a class in the package java.awt.

```
public void setDefaultCloseOperation(int operation)
```

Sets the action that will happen by default when the user clicks the close-window button. The argument should be one of the following defined constants:

JFrame.DO_NOTHING_ON_CLOSE: Do nothing. The JFrame does nothing, but if there are any registered window listeners, they are invoked. (Window listeners are explained in Chapter 18.)

JFrame.HIDE_ON_CLOSE: Hide the frame after invoking any registered WindowListener objects.

JFrame.DISPOSE_ON_CLOSE: Hide and *dispose* the frame after invoking any registered window listeners. When a window is **disposed** it is eliminated but the program does not end. To end the programs, you use the next constant as an argument to setDefaultCloseOperation.

JFrame.EXIT_ON_CLOSE: Exit the application using the System exit method. (Do not use this for frames in applets. Applets are discussed in Chapter 17.)

If no action is specified using the method `setDefaultCloseOperation`, then the default action taken is `JFrame.HIDE_ON_CLOSE`.

Throws:

`IllegalArgumentException` if the argument is not one of the values listed above.

`SecurityException` if the argument is `JFrame.EXIT_ON_CLOSE` and the Security Manager will not allow the caller to invoke `System.exit`. (You are not likely to encounter this case.)

```
public void setSize(int width, int height)
```

Sets the size of the calling frame so that it has the `width` and `height` specified. Pixels are the units of length used.

```
public void setTitle(String title)
```

Sets the title for this frame to the argument string.

To add a component to the `JFrame` use

```
getContentPane().add(Component componentAdded)
```

You do not use the `add` method directly on a `JFrame`, but instead use `add` with the content pane of the `JFrame`. Use of the `add` method with a `JFrame` calling object will produce a run-time error.

To set the layout manager use

```
getContentPane().setLayout(LayoutManager manager)
```

Layout managers are discussed in Chapter 16.

```
public void dispose()
```

Eliminates the calling frame and all its subcomponents. Any memory they use is released for reuse. If there are items left (items other than the calling frame and its subcomponents), then this does not end the program. (The method `dispose` is discussed in Chapter 18.)

```
public void setJMenuBar(JMenuBar menubar)
```

Sets the menu bar for the calling frame. (Menus and menu bars are discussed later in this chapter.)

`JMenuItem`

See `AbstractButton`.

`JOptionPane`

See Section 2.2 in Chapter 2.

JScrollPane

Package: javax.swing

Ancestor classes:

```
Object
   |
   +--Component
         |
         +--Container
               |
               +--JComponent

                     |
                     +--ScrollPane
```

Implements Interfaces: Accessible, ImageObserver, MenuContainer, ScrollPaneConstants, Serializable

```
public JScrollPane(Component objectToBeScrolled)
```

Creates a new JScrollPane for the objectToBeScrolled. Note that the objectToBeScrolled need not be a JTextArea, although that is the only type of argument considered in this book.

```
public void setHorizontalScrollBarPolicy(int policy)
```

Sets the policy for showing the horizontal scroll bar. The policy should be one of

JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS
JScrollPane.HORIZONTAL_SCROLLBAR_NEVER
JScrollPane.HORIZONTAL_SCROLLBAR_AS_NEEDED

The phrase AS_NEEDED means the scroll bar is shown only when it is needed. This is explained more fully in Chapter 18. The meanings of the other policy constants are obvious from their names.

(As indicated, these constants are defined in the class JScrollPane. You should not need to even be aware of the fact that they have int values. Think of them as policies, not as int values.)

```
public void setVerticalScrollBarPolicy(int policy)
```

Sets the policy for showing the vertical scroll bar. The policy should be one of

JScrollPane.VERTICAL_SCROLLBAR_ALWAYS
JScrollPane.VERTICAL_SCROLLBAR_NEVER
JScrollPane.VERTICAL_SCROLLBAR_AS_NEEDED

The phrase AS_NEEDED means the scroll bar is shown only when it is needed. This is explained more fully in Chapter 18. The meanings of the other policy constants are obvious from their names.

(As indicated, these constants are defined in the class JScrollPane. You should not need to even be aware of the fact that they have int values. Think of them as policies, not as int values.)

JTextArea

See JTextComponent.

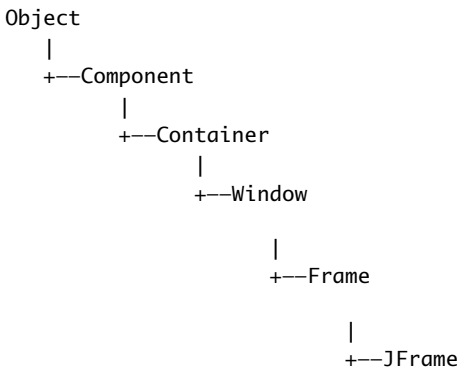
JTextComponent

Package: `javax.swing.text`

The classes `JTextField` and `JTextArea` are in the package `javax.swing`.

All these methods are inherited by the classes `JTextField` and `JTextArea`.

Ancestor classes:



Implements Interfaces: `Accessible, ImageObserver, MenuContainer, Scrollable, Serializable`

```
public String getText()
```

Returns the text that is displayed by this text component.

```
public boolean isEditable()
```

Returns `true` if the user can write in this text component. Returns `false` if the user is not allowed to write in this text component.

```
public void setBackground(Color theColor)
```

Sets the background color of this text component.

```
public void setEditable(boolean argument)
```

If `argument` is `true`, then the user is allowed to write in the text component. If `argument` is `false`, then the user is not allowed to write in the text component.

```
public void setText(String text)
```

Sets the text that is displayed by this text component to be the specified `text`.

JTextField

See `JTextComponent`.

List INTERFACE

package: java.util

The List interface extends the Collection interface.

All the exception classes mentioned are the kind that are not required to be caught in a catch block or declared in a throws clause.

All the exception classes mentioned are in the package java.lang and so do not require any import statement.

CONSTRUCTORS

Although not officially required by the interface, any class that implements the List interface should have at least two constructors: a no-argument constructor that creates an empty List object, and a constructor with one parameter of type Collection that creates a List object with the same elements as the constructor argument. If the argument imposes an ordering on its elements, then the List created should preserve this ordering.

```
public boolean contains(Object target)
```

Returns true if the calling object contains at least one instance of target. Uses target.equals to determine if target is in the calling object.

Throws:

ClassCastException if the type of target is incompatible with the calling object (optional).

NullPointerException if target is null and the calling object does not support null elements (optional).

```
public boolean containsAll(Collection collectionOfTargets)
```

Returns true if the calling object contains all of the elements in collectionOfTargets. For element in collectionOfTargets, uses element.equals to determine if element is in the calling object. The elements need not be in the same order or have the same multiplicity in collectionOfTargets and in the calling object.

```
public boolean equals(Object other)
```

If the argument is a List, returns true if the calling object and the argument contain exactly the same elements in exactly the same order; otherwise returns false. If the argument is not a List, false is returned.

```
public int hashCode()
```

Returns the hash code value for the calling object. Neither hash codes nor this method is discussed in this book. This entry is only here to make the definition of the List interface complete. You can safely ignore this entry until you go on to study hash codes in a more advanced book. In the meantime, if you need to implement this method, have it throw an UnsupportedOperationException.

```
boolean isEmpty()
```

Returns true if the calling object is empty; otherwise returns false.

`Iterator iterator()`

Returns an iterator for the calling object. (Iterators are discussed in Section 15.3.)

`public Object[] toArray()`

Returns an array containing all of the elements in the calling object. The elements in the returned array are in the same order as in the calling object. A new array must be returned so that the calling object has no references to the returned array.

`public Object[] toArray(Object[] a)`

Returns an array containing all of the elements in the calling object. The elements in the returned array are in the same order as in the calling object. The argument *a* is used primarily to specify the type of the array returned. The exact details are described in the table for the `Collection` interface.

Throws:

`ArrayStoreException` if the type of *a* is not an ancestor type of the type of every element in the calling object.

`NullPointerException` if *a* is `null`.

`public int size()`

Returns the number of elements in the calling object. If the calling object contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

Optional Methods

As with the `Collection` interface, the following methods are optional, which means they still must be implemented, but the implementation can simply throw an `UnsupportedOperationException` if for some reason you do not want to give them a "real" implementation. An `UnsupportedOperationException` is a `RunTimeException` and so is not required to be caught or declared in a `throws` clause.

`public boolean add(Object element)` *(Optional)*

Adds `element` to the end of the calling object's list. Normally returns `true`. Returns `false` if the operation failed, but if the operation failed, something is seriously wrong and you will probably get a run-time error anyway.

Throws:

`UnsupportedOperationException` if the *add* method is not supported by the calling object.

`ClassCastException` if the class of `element` prevents it from being added to the calling object.

`NullPointerException` if `element` is `null` and the calling object does not support `null` elements.

`IllegalArgumentException` if some aspect of `element` prevents it from being added to the calling object.

`IndexOutOfBoundsException` if the index does not satisfy:

`0 <= index <= size()`

`public boolean addAll(Collection collectionToAdd)` *(Optional)*

Adds all of the elements in `collectionToAdd` to the end of the calling object's list. The elements are added in the order they are produced by an iterator for `collectionToAdd`.

Throws:

`UnsupportedOperationException` if the `addAll` method is not supported by the calling object.

`ClassCastException` if the class of an element in `collectionToAdd` prevents it from being added to the calling object.

`NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.

`IllegalArgumentException` if some aspect of an element in `collectionToAdd` prevents it from being added to the calling object.

`public void clear()` *(Optional)*

Removes all the elements from the calling object.

Throws:

`UnsupportedOperationException` if the `clear` method is not supported by the calling object.

`public boolean remove(Object element)` *(Optional)*

Removes the first occurrence of `element` from the calling object's list, if it is present. Returns `true` if the calling object contained the `element`; returns `false` otherwise.

Throws:

`ClassCastException` if the type of `element` is incompatible with the calling object (optional).

`NullPointerException` if `element` is `null` and the calling object does not support `null` elements (optional).

`UnsupportedOperationException` if the `remove` method is not supported by the calling object.

`public boolean removeAll(Collection collectionToRemove)` *(Optional)*

Removes all the calling object's elements that are also in `collectionToRemove`. Returns `true` if the calling object was changed; otherwise returns `false`.

Throws:

`UnsupportedOperationException` if the `removeAll` method is not supported by the calling object.

`ClassCastException` if the types of one or more elements in `the calling object` are incompatible with `collectionToRemove` (optional).

`NullPointerException` if the calling object contains one or more `null` elements and `collectionToRemove` does not support `null` elements (optional).

`NullPointerException` if `collectionToRemove` is `null`.

`public boolean retainAll(Collection saveElements)` *(Optional)*

Retains only the elements in the calling object that are also in the collection `saveElements`. In other words, removes from the calling object all of its elements that are not contained in the collection `saveElements`. Returns `true` if the calling object was changed; otherwise returns `false`.

Throws:

`UnsupportedOperationException` if the `retainAll` method is not supported by the calling object.

`ClassCastException` if the types of one or more elements in the calling object are incompatible with `saveElements` (optional).

`NullPointerException` if the calling object contains one or more `null` elements and `saveElements` does not support `null` elements (optional).

`NullPointerException` if the collection `saveElements` is `null`.

New Method Headings

The following methods are in the `List` interface but were not in the `Collection` interface. Those that are optional are noted.

`public void add(int index, Object newElement)` *(Optional)*

Inserts `newElement` in the calling object's list at location `index`. The old elements at location `index` and higher are moved to higher indices. .

Throws:

`IndexOutOfBoundsException` if the `index` is not in the range:
`0 <= index <= size()`.

`UnsupportedOperationException` if this `add` method is not supported by the calling object.

`ClassCastException` if the class of `newElement` prevents it from being added to the calling object.

`NullPointerException` if `newElement` is `null` and the calling object does not support `null` elements.

`IllegalArgumentException` if some aspect of `newElement` prevents it from being added to the calling object.

`public boolean addAll(int index, Collection collectionToAdd)` *(Optional)*

Inserts all of the elements in `collectionToAdd` to the calling object's list starting at location `index`. The old elements at location `index` and higher are moved to higher indices. The elements are added in the order they are produced by an iterator for `collectionToAdd`. Returns `true` if successful; otherwise return `false`.

Throws:

`IndexOutOfBoundsException` if the `index` is not in the range:
`0 <= index <= size()`.

`UnsupportedOperationException` if the `addAll` method is not supported by the calling object.

`ClassCastException` if the class of one of the elements of `collectionToAdd` prevents it from being added to the calling object.

`NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.

`IllegalArgumentException` if some aspect of one of the elements of `collectionToAdd` prevents it from being added to the calling object.

`public int indexOf(Object target)`

Returns the index of the first element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns `-1` if `target` is not found.

Throws:

`ClassCastException` if the type of `target` is incompatible with the calling object (optional).

`NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

`public int lastIndexOf(Object target)`

Returns the index of the last element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns `-1` if `target` is not found.

Throws:

`ClassCastException` if the type of `target` is incompatible with the calling object (optional).

`NullPointerException` if `target` is `null` and the calling object does not support `null` elements (optional).

`public List subList(int fromIndex, int toIndex)`

Returns a *view* of the elements at locations `fromIndex` to `toIndex` of the calling object; the object at `fromIndex` is included; the object, if any, at `toIndex` is not included. The *view* uses references into the calling object; so changing the view can change the calling object. The returned object will be of type `List` but need not be of the same type as the calling object. Returns an empty `List` if `fromIndex` equals `toIndex`.

Throws:

`IndexOutOfBoundsException` if `fromIndex` and `toIndex` do not satisfy:

`0 <= fromIndex <= toIndex <= size()`.

`ListIterator listIterator()`

Returns a list iterator for the calling object. (Iterators are discussed in Section 15.3.)

`ListIterator listIterator(int index)`

Returns a list iterator for the calling object starting at `index`. The first element to be returned by the iterator is the one at `index`. (Iterators are discussed in Section 15.3.)

Throws:

`IndexOutOfBoundsException` if `index` does not satisfy:

`0 <= index <= size()`

`public Object get(int index)`

Returns the object at position `index`.

Throws an `IndexOutOfBoundsException` if the `index` is not in the range:

`0 <= index < size()`.

`public Object remove(int index)` *(Optional)*

Removes the element at position `index` in the calling object. Shifts any subsequent elements to the left (subtracts one from their indices). Returns the element that was removed from the calling object.

Throws:

`UnsupportedOperationException` if the `remove` method is not supported by the calling object.

`IndexOutOfBoundsException` if `index` does not satisfy:

```
0 <= index < size()
```

`public Object set(int index, Object newElement)` *(Optional)*

Sets the element at the specified `index` to `newElement`. The element previously at that position is returned.

Throws:

`IndexOutOfBoundsException` if the `index` is not in the range:
`0 <= index < size()`.

`UnsupportedOperationException` if the `set` method is not supported by the calling object.

`ClassCastException` if the class of `newElement` prevents it from being added to the calling object.

`NullPointerException` if `newElement` is `null` and the calling object does not support `null` elements.

`IllegalArgumentException` if some aspect of `newElement` prevents it from being added to the calling object.

`ListIterator` INTERFACE

package: `java.util`

The *cursor position* is explained in Chapter 15.

All the exception classes mentioned are the kind that are not required to be caught in a `catch` block or declared in a `throws` clause.

`NoSuchElementException` is in the `java.util` package, which requires an `import` statement if your code mentions the `NoSuchElementException` class. All the other exception classes mentioned are in the package `java.lang` and so do not require any `import` statements.

`public void add(Object newElement)` *(Optional)*

Inserts `newElement` at the location of the iterator cursor (that is, before the value, if any, that would be returned by `next()` and after the value, if any, that would be returned by `previous()`.)

Cannot be used if there has been a call to `add` or `remove` since the last call to `next()` or `previous()`.

Throws:

`IllegalStateException` if neither `next()` nor `previous()` has been called, or the `add` or `remove` method has already been called after the last call to `next()` or `previous()`.

`UnsupportedOperationException` if the `remove` operation is not supported by this `Iterator`.

`ClassCastException` if the class of `newElement` prevents it from being added.

`IllegalArgumentException` if some property other than the class of `newElement` prevents it from being added.

`public boolean hasNext()`

Returns `true` if there is a suitable element for `next()` to return; returns `false` otherwise.

`public boolean hasPrevious()`

Returns `true` if there is a suitable element for `previous()` to return; returns `false` otherwise.

`public int nextIndex()`

Returns the index of the element that would be returned by a call to `next()`. Returns the list size if the cursor position is at the end of the list.

`public Object next()`

Returns the next element of the list that produced the iterator. More specifically, returns the element immediately after the cursor position.

Throws:

`NoSuchElementException` if there is no next element.

`public Object previous()`

Returns the previous element of the list that produced the iterator. More specifically, returns the element immediately before the cursor position.

Throws:

`NoSuchElementException` if there is no previous element.

`public int previousIndex()`

Returns the index that would be returned by a call to `previous()`. Returns –1 if the cursor position is at the beginning of the list.

`public void remove()` *(Optional)*

Removes from the collection the last element returned by `next()` or `previous()`.

This method can be called only once per call to `next()` or `previous()`.
Cannot be used if there has been a call to `add` or `remove` since the last call to `next()` or `previous()`.

Throws:

`IllegalStateException` if neither `next()` nor `previous()` has been called, or the `add` or `remove` method has already been called after the last call to `next()` or `previous()`.

`UnsupportedOperationException` if the `remove` operation is not supported by this `Iterator`.

`public void set(Object newElement)` *(Optional)*

Replaces the last element returned by `next()` or `previous()` with `newElement`.

Cannot be used if there has been a call to `add` or `remove` since the last call to `next()` or `previous()`.

Throws:

`UnsupportedOperationException` if the `set` operation is not supported by this `Iterator`.

`IllegalStateException` if neither `next()` nor `previous()` has been called, or the `add` or `remove` method has been called since the last call to `next()` or `previous()`.

`ClassCastException` if the class of `newElement` prevents it from being added.

`IllegalArgumentException` if some property other than the class of `newElement` prevents it from being added.

Long

See Section 5.1 in Chapter 5.

Math

Package: `java.lang`

```
Object
   |
   +--Math
```

The `Math` class is marked `final`, which means it cannot be used as a base class to derive other classes.

```
public static double abs(double argument)
public static float abs(float argument)
public static long abs(long argument)
public static int abs(int argument)
```

Returns the absolute value of the `argument`. (The method name `abs` is overloaded to produce four similar methods.)

EXAMPLES

`Math.abs(-6)` and `Math.abs(6)` both return 6. `Math.abs(-5.5)` and `Math.abs(5.5)` both return `5.5`.

```
public static double ceil(double argument)
```

Returns the smallest whole number greater than or equal to the `argument`.

EXAMPLE:

`Math.ceil(3.2)` and `Math.ceil(3.9)` both return `4.0`.

```
public static double floor(double argument)
```

Returns the largest whole number less than or equal to the `argument`.

EXAMPLE:

`Math.floor(3.2)` and `Math.floor(3.9)` both return `3.0`.

```
public static double max(double n1, double n2)
public static float max(float n1, float n2)
public static long max(long n1, long n2)
public static int max(int n1, int n2)
```

Returns the maximum of the arguments `n1` and `n2`. (The method name `max` is overloaded to produce four similar methods.)

EXAMPLE:

`Math.max(3, 2)` returns 3.

```
public static double min(double n1, double n2)
public static float min(float n1, float n2)
public static long min(long n1, long n2)
public static int min(int n1, int n2)
```

Returns the minimum of the arguments n1 and n2. (The method name min is overloaded to produce four similar methods.)

EXAMPLE:

```
Math.min(3, 2) returns 2.
```

```
public static double pow(double base, double exponent)
```

Returns base to the power exponent.

EXAMPLE:

Math.pow(2.0,3.0) returns 8.0.

```
public static long round(double argument)
public static int round(float argument)
```

Rounds its argument.

EXAMPLES

Math.round(3.2) returns 3. Math.round(3.6) returns 4.

```
public static double sqrt(double argument)
```

Returns the square root of its argument.

EXAMPLE:

```
Math.sqrt(4) returns 2.0.
```

ObjectInputStream

Package: java.io

The FileInputStream class is also in this package.

Ancestor classes:

```
Object
   |
   +--InputStream
         |
         +--ObjectInputStream
```

Implements Interfaces:

DataInput, ObjectInput, ObjectStreamConstants

```
public ObjectInputStream(InputStream streamObject)
```

There is no constructor that takes a file name as an argument. If you want to create a stream using a file name, you use

```
new ObjectInputStream(new FileInputStream(File_Name))
```

Alternatively, you can use an object of the class `File` in place of the *File_Name*, as follows:

```
new ObjectInputStream(new FileInputStream(File_Object))
```

The constructor for `FileInputStream` may throw a `FileNotFoundException`, which is a kind of `IOException`. If the `FileInputStream` constructor succeeds, then the constructor for `ObjectInputStream` may throw a different `IOException`.

```
public void close() throws IOException
```

Closes the stream's connection to a file.

```
public boolean readBoolean() throws IOException
```

Reads a `boolean` value from the input stream and returns that `boolean` value. If `readBoolean` tries to read a value from the file and that value was not written using the method `writeBoolean` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public char readChar() throws IOException
```

Reads a `char` value from the input stream and returns that `char` value. If `readChar` tries to read a value from the file and that value was not written using the method `writeChar` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public double readDouble() throws IOException
```

Reads a `double` value from the input stream and returns that `double` value. If `readDouble` tries to read a value from the file and that value was not written using the method `writeDouble` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public float readFloat() throws IOException
```

Reads a `float` value from the input stream and returns that `float` value. If `readFloat` tries to read a value from the file and that value was not written using the method `writeFloat` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public int readInt() throws IOException
```

Reads an `int` value from the input stream and returns that `int` value. If `readInt` tries to read a value from the file and that value was not written using the method `writeInt` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public long readLong() throws IOException
```

Reads a `long` value from the input stream and returns that `long` value. If `readLong` tries to read a value from the file and that value was not written using the method `writeLong` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
Object readObject( ) throws ClassNotFoundException, IOException
```

Reads an object from the input stream. The object read should have been written using `writeObject` of the class `ObjectOutputStream`.

Throws:

`ClassNotFoundException` if the class of a serialized object cannot be found. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown. May throw various other `IOExceptions`.

```
public int readShort() throws IOException
```

Reads a `short` value from the input stream and returns that `short` value. If `readInt` tries to read a value from the file and that value was not written using the method `writeShort` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public String readUTF() throws IOException
```

Reads a `String` value from the input stream and returns that `String` value. If `readUTF` tries to read a value from the file and that value was not written using the method `writeUTF` of the class `ObjectOutputStream` (or written in some equivalent way), then problems will occur. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public int skipBytes(int n) throws IOException
```

Skips `n` bytes.

ObjectOutputStream

Package: `java.io`

The `FileOutputStream` class is also in this package.

Ancestor classes:

```
Object
   |
   +--OutputStream
         |
         +--ObjectOutputStream
```

Implements Interfaces:

`DataOutput, ObjectOutput, ObjectStreamConstants`

```
public ObjectOutputStream(OutputStream streamObject)
```

There is no constructor that takes a file name as an argument. If you want to create a stream using a file name, you use

```
new ObjectOutputStream(new FileOutputStream(File_Name))
```

This creates a blank file. If there already is a file named *File_Name*, then the old contents of the file are lost.

If you want to create a stream using an object of the class `File`, you use

```
new ObjectOutputStream(new FileOutputStream(File_Object))
```

The constructor for `FileOutputStream` may throw a `FileNotFoundException`, which is a kind of `IOException`. If the `FileOutputStream` constructor succeeds, then the constructor for `ObjectOutputStream` may throw a different `IOException`.

```
public void close() throws IOException
```

Closes the stream's connection to a file. This method calls `flush` before closing the file.

```
public void flush() throws IOException
```

Flushes the output stream. This forces an actual physical write to the file of any data that has been buffered and not yet physically written to the file. Normally, you should not need to invoke `flush`.

```
public void writeBoolean(boolean b) throws IOException
```

Writes the `boolean` value `b` to the output stream.

```
public void writeChar(int n) throws IOException
```

Writes the `char` value `n` to the output stream. Note that it expects its argument to be an `int` value. However, if you simply use the `char` value, then Java will automatically type cast it to an `int` value. The following are equivalent:

```
outputStream.writeChar((int)'A');
```

and

```
outputStream.writeChar('A');
```

```
public void writeDouble(double x) throws IOException
```

Writes the `double` value `x` to the output stream.

```
public void writeFloat(float x) throws IOException
```

Writes the `float` value `x` to the output stream.

```
public void writeInt(int n) throws IOException
```

Writes the `int` value `n` to the output stream.

```
public void writeLong(long n) throws IOException
```

Writes the `long` value `n` to the output stream.

```
public void writeObject(Object anObject) throws IOException
```

Writes its argument to the output stream. The object argument should be an object of a serializable class, a concept discussed in Chapter 10. Throws various `IOExceptions`.

```
public void writeShort(short n) throws IOException
```

Writes the `short` value `n` to the output stream.

```
public void writeUTF(String aString) throws IOException
```

Writes the `String` value `aString` to the output stream. UTF refers to a particular method of encoding the string. To read the string back from the file, you should use the method `readUTF` of the class `ObjectInputStream`.

PrintWriter

Package: `java.io`

The `FileOutputStream` class is also in this package.

Ancestor classes:

```
Object
   |
   +--Writer
         |
         +--PrintWriter
```

```
public PrintWriter(OutputStream streamObject)
```

This is the only constructor you are likely to need. There is no constructor that accepts a file name as an argument. If you want to create a stream using a file name, you use

```
new PrintWriter(new FileOutputStream(File_Name))
```

When the constructor is used in this way, a blank file is created. If there already was a file named *File_Name*, then the old contents of the file are lost. If you want instead to append new text to the end of the old file contents, use

```
new PrintWriter(new FileOutputStream(File_Name, true))
```

(For an explanation of the argument `true`, see Chapter 10.)

When used in either of these ways, the `FileOutputStream` constructor, and so the `PrintWriter` constructor invocation, can throw a `FileNotFoundException`, which is a kind of `IOException`.

If you want to create a stream using an object of the class `File`, you can use a `File` object in place of the *File_Name*.

```
public void close()
```

Closes the stream's connection to a file. This method calls `flush` before closing the file.

`public void flush()`

Flushes the output stream. This forces an actual physical write to the file of any data that has been buffered and not yet physically written to the file. Normally, you should not need to invoke `flush`.

`public final void print(`*Argument*`)`

Same as `println`, except that this method does not end the line, and so the next output will be on the same line.

`public final void println(`*Argument*`)`

The *Argument* can be a string, character, integer, floating-point number, boolean value, or any combination of these, connected with + signs. The *Argument* can also be any object, although it will not work as desired unless the object has a properly defined `toString()` method. The *Argument* is output to the file connected to the stream. After the *Argument* has been output, the line ends, and so the next output is sent to the next line.

RandomAccessFile

Package: `java.io`

```
Object
   |
   +--RandomAccessFile
```

Implements Interfaces: `DataInput, DataOutput`

`public RandomAccessFile(String fileName, String mode)`

`public RandomAccessFile(File fileObject, String mode)`

Opens the file, does not delete data already in the file, but does position the file pointer at the first (zeroth) location.

The `mode` must be one of the following:

`"r"` Open for reading only.

`"rw"` Open for reading and writing.

`"rws"` Same as `"rw"`, and also requires that every update to the file's content or metadata be written synchronously to the underlying storage device.

`"rwd"` Same as `"rw"`, and also requires that every update to the file's content be written synchronously to the underlying storage device.

`"rws"` and `"rwd"` are not covered in this book.

`public void close() throws IOException`

Closes the stream's connection to a file.

```
public void setLength(long newLength) throws IOException
```

Sets the length of this file.

If the present length of the file as returned by the `length` method is greater than the `newLength` argument, then the file will be truncated. In this case, if the file pointer location as returned by the `getFilePointer` method is greater than `newLength`, then after this method returns, the file pointer location will be equal to `newLength`.

If the present length of the file as returned by the `length` method is smaller than `newLength`, then the file will be extended. In this case, the contents of the extended portion of the file are not defined.

```
public long getFilePointer() throws IOException
```

Returns the current location of the file pointer. Locations are numbered starting with 0.

```
public long length() throws IOException
```

Returns the length of the file.

```
public int read() throws IOException
```

Reads a byte of data from the file and returns it as an integer in the range 0 to 255.

```
public int read(byte[] a) throws IOException
```

Reads up to `a.length` bytes of data from the file into the array of bytes. Returns the total number of bytes read or -1 if the end of the file is reached.

```
public final boolean readBoolean() throws IOException
```

Reads a `boolean` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public final byte readByte() throws IOException
```

Reads a `byte` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public final char readChar() throws IOException
```

Reads a `char` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public final double readDouble() throws IOException
```

Reads a `double` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public final float readFloat() throws IOException
```

Reads a `float` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public final int readInt() throws IOException
```

Reads an `int` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public final long readLong() throws IOException
```

Reads a `long` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public final short readShort() throws IOException
```

Reads a `short` value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public final String readUTF() throws IOException
```

Reads a String value from the file and returns that value. If an attempt is made to read beyond the end of the file, an `EOFException` is thrown.

```
public void seek(long location) throws IOException
```

Moves the file pointer to the specified `location`.

```
public void write(byte[] a) throws IOException
```

Writes `a.length` bytes from the specified byte array to the file.

```
public void write(int b) throws IOException
```

Writes the specified byte to the file.

```
public final void writeBoolean(boolean b) throws IOException
```

Writes the `boolean` b to the file.

```
public final void writeByte(byte b) throws IOException
```

Writes the `byte` b to the file.

```
public final void writeChar(char c) throws IOException
```

Writes the `char` c to the file.

```
public final void writeDouble(double d) throws IOException
```

Writes the `double` d to the file.

```
public final void writeFloat(float f) throws IOException
```

Writes the `float` f to the file.

```
public final void writeInt(int n) throws IOException
```

Writes the `int` n to the file.

```
public final void writeLong(long n) throws IOException
```

Writes the `long` n to the file.

```
public final void writeShort(short n) throws IOException
```

Writes the short n to the file.

```
public final void writeUTF(String s) throws IOException
```

Writes the String s to the file.

Serializable INTERFACE

See Section 10.4 in Chapter 10.

Set INTERFACE

package: java.util

The Set interface extends the Collection interface.

All the exception classes mentioned are the kind that are not required to be caught in a catch block or declared in a throws clause.

All the exception classes mentioned are in the package java.lang and so do not require any import statement.

CONSTRUCTORS

Although not officially required by the interface, any class that implements the Set interface should have at least two constructors: a no-argument constructor that creates an empty Set object, and a constructor with one parameter of type Collection that creates a Set object with the same elements as the constructor argument.

```
public boolean contains(Object target)
```

Returns true if the calling object contains at least one instance of target. Uses target.equals to determine if target is in the calling object.

Throws:

ClassCastException if the type of target is incompatible with the calling object (optional).

NullPointerException if target is null and the calling object does not support null elements (optional).

```
public boolean containsAll(Collection collectionOfTargets)
```

Returns true if the calling object contains all of the elements in collectionOfTargets. For element in collectionOfTargets, this method uses element.equals to determine if element is in the calling object. If collectionOfTargets is itself a Set, this is a test to see if collectionOfTargets is a subset of the calling object.

Throws:

ClassCastException if the types of one or more elements in collectionOfTargets are incompatible with the calling object (optional).

NullPointerException if collectionOfTargets contains one or more null elements and the calling object does not support null elements (optional).

NullPointerException if collectionOfTargets is null.

`public boolean equals(Object other)`

If the argument is a `Set`, returns `true` if the calling object and the argument contain exactly the same elements; otherwise returns `false`. If the argument is not a `Set`, `false` is returned.

`public int hashCode()`

Returns the hash code value for the calling object. Neither hash codes nor this method is discussed in this book. This entry is only here to make the definition of the `Collection` interface complete. You can safely ignore this entry until you go on to study hash codes in a more advanced book. In the meantime, if you need to implement this method, have it throw an `UnsupportedOperationException`.

`boolean isEmpty()`

Returns `true` if the calling object is empty; otherwise returns `false`.

`Iterator iterator()`

Returns an iterator for the calling object. (Iterators are discussed in Section 15.3.)

`public Object[] toArray()`

Returns an array containing all of the elements in the calling object. A new array should be returned so that the calling object has no references to the returned array.

`public Object[] toArray(Object[] a)`

Returns an array containing all of the elements in the calling object. The argument `a` is used primarily to specify the type of the array returned. The exact details are described in the table for the `Collection` interface.

Throws:

`ArrayStoreException` if the type of `a` is not an ancestor type of the type of every element in the calling object.

`NullPointerException` if `a` is `null`.

`public int size()`

Returns the number of elements in the calling object. If the calling object contains more than `Integer.MAX_VALUE` elements, returns `Integer.MAX_VALUE`.

Adding and Removing Elements

Unlike the `Collection` interface, the following methods are almost always implemented for the `Set` interface.

`public boolean add(Object element)`

If `element` is not already in the calling object, `element` is added to the calling object and `true` is returned. If `element` is in the calling object, the calling object is unchanged and `false` is returned.

Throws:

`UnsupportedOperationException` if the `add` method is not supported by the set.

`ClassCastException` if the class of `element` prevents it from being added to the set.

`NullPointerException` if `element` is `null` and the set does not support `null` elements.

`IllegalArgumentException` if some other aspect of `element` prevents it from being added to this set.

`public boolean addAll(Collection collectionToAdd)`

Ensures that the calling object contains all the elements in `collectionToAdd`. Returns `true` if the calling object changed as a result of the call; returns `false` otherwise. Thus, if `collectionToAdd` is a `Set`, then the calling object is changed to the union of itself with `collectionToAdd`.

Throws:

`UnsupportedOperationException` if the `addAll` method is not supported by the set.

`ClassCastException` if the class of some element of `collectionToAdd` prevents it from being added to the calling object.

`NullPointerException` if `collectionToAdd` contains one or more `null` elements and the calling object does not support `null` elements, or if `collectionToAdd` is `null`.

`IllegalArgumentException` if some aspect of some element of `collectionToAdd` prevents it from being added to the calling object.

`public void clear()`

Removes all the elements from the calling object.

Throws:

`UnsupportedOperationException` if the `clear` method is not supported by the calling object.

`public boolean remove(Object element)`

Removes the `element` from the calling object, if it is present. Returns `true` if the calling object contained the `element`; returns `false` otherwise.

Throws:

`ClassCastException` if the type of `element` is incompatible with the calling object (optional).

`NullPointerException` if `element` is `null` and the calling object does not support `null` elements (optional).

`UnsupportedOperationException` if the `remove` method is not supported by the calling object.

`public boolean removeAll(Collection collectionToRemove)`

Removes all the calling object's elements that are also contained in `collectionToRemove`. Returns `true` if the calling object was changed; otherwise returns `false`.

Throws:

`UnsupportedOperationException` if the `removeAll` method is not supported by the calling object.

`ClassCastException` if the types of one or more elements in `collectionToRemove` are incompatible with the calling object (optional).

`NullPointerException` if the calling object contains a `null` element and `collectionToRemove` does not support `null` elements (optional).

`NullPointerException` if `collectionToRemove` is `null`.

```
public boolean retainAll(Collection saveElements)
```

Retains only the elements in the calling object that are also contained in the collection `saveElements`. In other words, removes from the calling object all of its elements that are not contained in the collection `saveElements`. Returns `true` if the calling object was changed; otherwise returns `false`. If the argument is itself a `Set`, this changes the calling object to the intersection of itself with the argument.

Throws:

`UnsupportedOperationException` if the `retainAll` method is not supported by the calling object.

`ClassCastException` if the types of one or more elements in the calling object are incompatible with `saveElements` (optional).

`NullPointerException` if `saveElements` contains a `null` element and the calling object does not support `null` elements (optional).

`NullPointerException` if `saveElements` is `null`.

Short

See Section 5.1 in Chapter 5.

String

Package: `java.lang`

`String` is marked `final` and so you cannot use it as a base class to derive another class.

Implements Interfaces: `CharSequence`, `Comparable`, `Serializable`

Ancestor classes:

```
Object
   |
   +--String
```

CONSTRUCTORS

```
public String()
```

Creates a `String` object that represents an empty character sequence. Note that this is a pretty useless constructor since `String` objects are immutable.

```
public String(BufferedString buffer)
```

Creates a new `String` object that contains the same sequence of characters that is currently contained in the `BufferedString` argument. This is a deep copy; subsequent modification of the `BufferedString` object does not affect the newly created string.

Throws:

`NullPointerException` if `buffer` is `null`.

```
public String(char[] value, int offset, int count)
```

Creates a new `String` that contains characters from a subarray of the character array argument. The `offset` argument is the index of the first character of the subarray, and the `count` argument specifies the length of the subarray. The contents of the subarray are copied. This is a deep copy; subsequent modifications of the character array do not affect the newly created string.

Throws:

`IndexOutOfBoundsException` if the elements specified by `offset` and `count` are not all within the bounds of the `value` array.

`NullPointerException` if `value` is `null`.

```
public String(String original)
```

Creates a new `String` object so that it represents the same sequence of characters as the argument. Unless an explicit copy of `original` is needed, use of this constructor is unnecessary since `String` objects are immutable.

Throws:

`NullPointerException` if `original` is `null`.

Methods

```
public char charAt(int position)
```

Returns the character in the calling object string at the `position`. Positions are counted 0, 1, 2, etc.

Example:

After program executes `String greeting = "Hello!";`

`greeting.charAt(0)` returns `'H'`, and

`greeting.charAt(1)` returns `'e'`.

Throws:

`IndexOutOfBoundsException` if `position` is negative or not less than the length of the calling object string.

```
public int compareTo(String aString)
```

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering. Lexicographic order is the same as alphabetical order but with the characters ordered as in Appendix 3. Note that in Appendix 3 all the uppercase letters are in regular alphabetical order and all the lowercase letters are in alphabetical order, but all the uppercase letters precede all the lowercase letters. So, lexicographic ordering is the same as alphabetical ordering when either both strings are all uppercase letters or both strings are all lowercase letters. If the calling string is first, it returns a negative value. If the two strings are equal, it returns zero. If the argument is first, it returns a positive number.

Example:

After program executes `String entry = "adventure";`

`entry.compareTo("zoo")` returns a negative number,

`entry.compareTo("adventure")` returns `0`, and

`entry.compareTo("above")` returns a positive number.

Throws:

`NullPointerException` if `aString` is `null`.

`public int compareToIgnoreCase(String aString)`

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering, treating uppercase and lowercase letters as being the same. (To be precise, all uppercase letters are treated as if they were their lowercase versions in doing the comparison.) Thus, if both strings consist entirely of letters, the comparison is for ordinary alphabetical order. If the calling string is first, it returns a negative value. If the two strings are equal, ignoring cases, it returns zero. If the argument is first, it returns a positive number.

EXAMPLE:

After program executes `String entry = "adventure";`

`entry.compareToIgnoreCase("Zoo")` returns a negative number,

`entry.compareToIgnoreCase("Adventure")` returns `0`, and

`"Zoo".compareToIgnoreCase(entry)` returns a positive number.

Throws:

`NullPointerException` if `aString` is `null`.

`public boolean contentEquals(StringBuffer stringBufferObject)`

Returns `true` if and only if this `String` represents the same sequence of characters as the `StringBuffer` argument.

Throws:

`NullPointerException` if `stringBufferObject` is `null`.

`public boolean equals(String otherString)`

Returns `true` if the calling object string and the `otherString` are equal. Otherwise returns `false`.

EXAMPLE:

After program executes `String greeting = "Hello";`

`greeting.equals("Hello")` returns `true`

`greeting.equals("Good-Bye")` returns `false`

`greeting.equals("hello")` returns `false`

Note that case matters: `"Hello"` and `"hello"` are not equal because one starts with an uppercase letter and the other starts with a lowercase letter.

`public boolean equalsIgnoreCase(String otherString)`

Returns `true` if the calling object string and the `otherString` are equal, considering uppercase and lowercase versions of a letter to be the same. Otherwise returns `false`.

EXAMPLE:

After program executes `String name = "mary";`

`name.equalsIgnoreCase("Mary")` returns `true`

`public int indexOf(String aString)`

Returns the index (position) of the first occurrence of the string `aString` in the calling object string. Positions are counted 0, 1, 2, etc. Returns `–1` if `aString` is not found.

EXAMPLE:

After program executes `String greeting = "Hi Mary!";`
`greeting.indexOf("Mary")` returns 3, and
`greeting.indexOf("Sally")` returns `–1`.

Throws:

`NullPointerException` if `aString` is `null`.

`public int indexOf(String aString, int start)`

Returns the index (position) of the first occurrence of the string `aString` in the calling object string that occurs at or after position `start`. Positions are counted 0, 1, 2, etc. Returns `–1` if `aString` is not found.

EXAMPLE:

After program executes `String name = "Mary, Mary quite contrary";`
`name.indexOf("Mary", 1)` returns 6.
The same value is returned if 1 is replaced by any number up to and including 6.
`name.indexOf("Mary", 0)` returns 0.
`name.indexOf("Mary", 8)` returns `–1`.

Throws:

`NullPointerException` if `aString` is `null`.

`public int lastIndexOf(String aString)`

Returns the index (position) of the last occurrence of the string `aString` in the calling object string. Positions are counted 0, 1, 2, etc. Returns `–1` if `aString` is not found.

EXAMPLE:

After program executes `String name = "Mary, Mary, Mary quite so";`
`greeting.indexOf("Mary")` returns 0, and
`name.lastIndexOf("Mary")` returns 12.

Throws:

`NullPointerException` if `aString` is `null`.

`public int length()`

Returns the length of the calling object (which is a string) as a value of type `int`.

EXAMPLE:

After program executes `String greeting = "Hello!";`
`greeting.length()` returns 6.

```
public String substring(int start)
```

Returns the substring of the calling object string starting from `start` through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position `start` is included in the value returned.

EXAMPLE:

After program executes `String sample = "AbcdefG";`
`sample.substring(2)` returns `"cdefG"`.

Throws:

`IndexOutOfBoundsException` if `start` is negative or larger than the length of the calling object.

```
public String substring(int start, int end)
```

Returns the substring of the calling object string starting from position `start` through, but not including, position `end` of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position `start` is included in the value returned, but the character at position `end` is not included.

EXAMPLE:

After program executes `String sample = "AbcdefG";`
`sample.substring(2, 5)` returns `"cde"`.

Throws:

`IndexOutOfBoundsException` if the `start` is negative, or `end` is larger than the length of this `String` object, or `start` is larger than `end`.

```
public String toLowerCase()
```

Returns a string with the same characters as the calling object string, but with all letter characters converted to lowercase.

EXAMPLE:

After program executes `String greeting = "Hi Mary!";`
`greeting.toLowerCase()` returns `"hi mary!"`.

```
public String toUpperCase()
```

Returns a string with the same characters as the calling object string, but with all letter characters converted to uppercase.

EXAMPLE:

After program executes `String greeting = "Hi Mary!";`
`greeting.toUpperCase()` returns `"HI MARY!"`.

```
public String trim()
```

Returns a string with the same characters as the calling object string, but with leading and trailing white space removed. Whitespace characters are the characters that print as white space on paper, such as the blank (space) character, the tab character, and the new-line character `'\n'`.

EXAMPLE:

After program executes `String pause = "    Hmm    ";`
`pause.trim()` returns `"Hmm"`.

StringBuffer

Package: `java.lang`

`StringBuffer` is marked `final` and so you cannot use it as a base class to derive another class.

Implements Interfaces: `CharSequence`, `Serializable`

Ancestor classes:

```
Object
   |
   +--StringBuffer
```

CONSTRUCTORS

```
public StringBuffer()
```

Creates a `StringBuffer` object with no characters in it and an initial capacity of 16 characters.

```
public StringBuffer(int capacity)
```

Constructs a `StringBuffer` object with no characters in it and an initial capacity specified by the argument.

Throws:

`NegativeArraySizeException` if `length` is less than 0. `NegativeArraySizeException` is a derived class of `RuntimeException`, and so is an unchecked exception, which means it is not required to be caught or declared in a `throws` clause.

```
public StringBuffer(String ordinaryString)
```

Constructs a string buffer so that it represents the same sequence of characters as the `ordinaryString` argument; in other words, the initial content of the string buffer is a copy of `ordinaryString`. The initial capacity of the string buffer is 16 plus the length of `ordinaryString`.

Throws:

`NullPointerException` if `ordinaryString` is `null`.

METHODS

```
public StringBuffer append(char[] charArray,
                           int offset, int length)
```

Appends the string representation of the characters in `charArray` starting at `charArray[offset]` and extending for a total of `length` characters. Note that the calling object is changed and a reference to the changed calling object is returned.

Throws:

`ArrayIndexOutOfBoundsException` if `offset` and `length` are not consistent with the range of `charArray`.

```
public StringBuffer append(char c)
```

Appends the character argument to the `StringBuffer` calling object and returns this longer string.

```
public StringBuffer append(char[] charArray)
```

Appends the string representation of the `char` array argument to this string buffer. Note that the calling object is changed and a reference to the changed calling object is returned.

```
public StringBuffer append(double d)
```

Appends the string representation of the `double` argument to the `StringBuffer` calling object and returns this longer string.

```
public StringBuffer append(float d)
```

Appends the string representation of the `float` argument to the `StringBuffer` calling object and returns this longer string.

```
public StringBuffer append(int n)
```

Appends the string representation of the `int` argument to the `StringBuffer` calling object and returns this longer string.

```
public StringBuffer append(long n)
```

Appends the string representation of the `long` argument to the `StringBuffer` calling object and returns this longer string.

```
public StringBuffer append(String ordinaryString)
```

Appends the `String` argument to the `StringBuffer` calling object and returns this longer string.

If `ordinaryString` is `null`, then the four characters `"null"` are appended to this string buffer.

Note that the calling object is changed and a reference to the changed calling object is returned.

```
public StringBuffer append(StringBuffer bufferedString)
```

Appends the `StringBuffer` argument to the `StringBuffer` calling object and returns this longer string.

If `bufferedString` is `null`, then the four characters `"null"` are appended to this string buffer.

Note that the calling object is changed and a reference to the changed calling object is returned.

`public int capacity()`

Returns the current capacity of the calling object. The capacity is the amount of storage currently available for characters. The capacity will automatically be increased if necessary.

`public char charAt(int position)`

Returns the character in the calling object string at `position`. Positions are counted 0, 1, 2, etc.

Throws:

`IndexOutOfBoundsException` if `position` is negative or not less than the length of the calling object.

`contentEquals`

There is no such method for the class `StringBuffer`, but see the method `contentEquals` for the class `String`.

`public StringBuffer delete(int start, int end)`

Removes the characters in a substring of the calling object. The substring to remove begins at the specified start and extends to the character at index `end` − 1 or to the end of the calling object if no such character exists. If `start` is equal to `end`, no changes are made. Note that the calling object is changed and a reference to the changed calling object is returned.

Throws:

`StringIndexOutOfBoundsException` if `start` is negative, greater than `length()`, or greater than `end`. `StringIndexOutOfBoundsException` is a derived class of `RuntimeException`, and so is an unchecked exception, which means it is not required to be caught or declared in a `throws` clause.

`public void ensureCapacity(int minimumCapacity)`

Ensures that the capacity of the calling object is at least equal to `minimumCapacity`. If the current capacity of the calling object is less than `minimumCapacity`, then the capacity is increased. The new capacity is the larger of: `minimumCapacity` and twice the old capacity, plus 2.

If the `minimumCapacity` is nonpositive, this method takes no action and simply returns.

`Start public boolean equals(Object otherObject)`

Warning: This is the method inherited from `Object`. It is not overridden for the class `StringBuffer` and does not work as you might expect. Normally, it should not be used.

`public int indexOf(String aString)`

Returns the index (position) of the first occurrence of the string `aString` in the calling object. Positions are counted 0, 1, 2, etc. Returns −1 if `aString` is not found. Note that the argument is of type `String`, not `StringBuffer`.

Throws:

`NullPointerException` if `aString` is `null`.

```
public int indexOf(String aString, int start)
```

Returns the index (position) of the first occurrence of the string `aString` in the calling object that occurs at or after position `start`. Positions are counted 0, 1, 2, etc. Returns –1 if `aString` is not found. Note that the argument is of type `String`, not `StringBuffer`.

Throws:

`NullPointerException` if `aString` is `null`.

```
public int lastIndexOf(String aString)
```

Returns the index (position) of the last occurrence of the string `aString` in the calling object string. Positions are counted 0, 1, 2, etc. Returns –1 if `aString` is not found. Note that the argument is of type `String`, not `StringBuffer`.

Throws:

`NullPointerException` if `aString` is `null`.

```
public int length()
```

Returns the length of the calling object as a value of type `int`.

```
public StringBuffer replace(int start,
                            int end, String ordinaryString)
```

Replaces the characters in a substring of the calling object with characters in the `ordinaryString`. The substring begins at the specified `start` and extends to the character at index `end` –1 or to the end of the calling object if no such character exists. First the characters in the substring are removed and then the specified `ordinaryString` is inserted at `start`. (The calling object will be lengthened to accommodate the `ordinaryString` if necessary.) Note that the calling object is changed and a reference to the changed calling object is returned.

Throws:

`StringIndexOutOfBoundsException` if `start` is negative, greater than `length()`, or greater than `end`. `StringIndexOutOfBoundsException` is a derived class of `RuntimeException`, and so is an unchecked exception, which means it is not required to be caught or declared in a `throws` clause.

```
public void setLength(int newLength)
```

Sets the length of the calling object. The calling object is altered to represent a new character sequence whose length is specified by the argument. For every nonnegative index *k* less than `newLength`, the character at index *k* in the new character sequence is the same as the character at index *k* in the old sequence. If the `newLength` argument is less than the current length of the string buffer, the string buffer is truncated to contain exactly the number of characters given by the `newLength` argument.

If the `newLength` argument is greater than the current length, sufficient null characters (`'\u0000'`) are appended to the string buffer so that length becomes the `newLength` argument.

Throws:

`IndexOutOfBoundsException` if `newLength` is negative.

```
public String substring(int start)
```

Returns the substring of the calling object starting from `start` through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position `start` is included in the value returned. Note that the substring is returned as a value of type `String`, not `StringBuffer`.

Throws:

`StringIndexOutOfBoundsException` if `start` is negative or larger than the length of the calling object. `StringIndexOutOfBoundsException` is a derived class of `RuntimeException`, and so is an unchecked exception, which means it is not required to be caught or declared in a `throws` clause.

```
public String substring(int start, int end)
```

Returns the substring of the calling object starting from position `start` through, but not including, position `end` of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position `start` is included in the value returned, but the character at position `end` is not included. Also note that the substring is returned as a value of type `String`, not `StringBuffer`.

Throws:

`StringIndexOutOfBoundsException` if `start` is negative, or `end` is larger than the length of this calling object, or `start` is larger than `end`. `StringIndexOutOfBoundsException` is a derived class of `RuntimeException`, and so is an unchecked exception, which means it is not required to be caught or declared in a `throws` clause.

```
public String toString()
```

Creates a new `String` object that contains the same character sequence calling object and returns that `String` object. Subsequent changes to the calling object do not affect the contents of the `String` returned.

StringTokenizer

The class `StringTokenizer` is in the `java.util` package.

Implements Interface: `Enumeration`

Ancestor classes:

```
Object
   |
   +--StringTokenizer
```

```
public StringTokenizer(String theString)
```

Constructor for a tokenizer that will use whitespace characters as separators when finding tokens in `theString`.

```
public StringTokenizer(String theString, String delimiters)
```

Constructor for a tokenizer that will use the characters in the string `delimiters` as separators when finding tokens in `theString`.

```
public StringTokenizer(String theString, String delimiters,
                       boolean returnDelimiters)
```

Creates a tokenizer similar to `StringTokenizer(String theString, String delimiters)`, but with the following differences: If `returnDelimiters` is `true`, the delimiters are also returned by `nextToken`; each delimiter is returned as a one-character `String`. If `returnDelimiters` is `false`, the delimiters are not returned by `nextToken`. Thus, if `returnDelimiters` is `false`, the tokenizer created is the same as with `StringTokenizer(String theString, String delimiters)`.

```
public int countTokens()
```

Returns the number of tokens remaining to be returned by `nextToken`.

```
public boolean hasMoreElements()
```

Same as `hasMoreTokens`.

```
public boolean hasMoreTokens()
```

Tests whether there are more tokens available from this tokenizer's string. When used in conjunction with `nextToken`, it returns `true` as long as `nextToken` has not yet returned all the tokens in the string; returns `false` otherwise.

```
public String nextToken()
```

Returns the next token from this tokenizer's string.

Throws:

`NoSuchElementException` if there are no more tokens to return. `NoSuchElementException` is one of the exceptions that need not be declared in a `throws` clause or caught in a `catch` block.

Vector

Package: `java.util`

Implements Interfaces: `Cloneable`, `Collection`, `List`, `RandomAccess`, `Serializable`

Ancestor classes:

```
Object
   |
   +--AbstractCollection
         |
         +--AbstractList
               |
               +--Vector
```

Constructors

```
public Vector( )
```

Creates an empty vector with an initial capacity of 10. When the vector needs to increase its capacity, the capacity doubles.

```
public Vector(Collection c)
```

Creates a vector that contains all the elements of the collection `c` in the same order as they have in `c`. If `c` is a vector, the capacity of the created vector will be `c.size()`, not `c.capacity`.

Throws:

`NullPointerException` if `c` is `null`.

```
public Vector(int initialCapacity)
```

Creates an empty vector with the specified initial capacity. When the vector needs to increase its capacity, the capacity doubles.

```
public Vector(int initialCapacity, int capacityIncrement)
```

Constructs an empty vector with the specified initial capacity and capacity increment. When the vector needs to grow, it will add room for `capacityIncrement` more items.

Arraylike Methods

```
public Object get(int index)
```

Returns the element at the specified index. This is analogous to returning `a[index]` for an array `a`.

Throws:

`ArrayIndexOutOfBoundsException` if the `index` is not greater than or equal to `0` and less than the current size of the vector.

```
public Object set(int index, Object newElement)
```

Sets the element at the specified `index` to `newElement`. The element previously at that position is returned. If you draw an analogy between the vector and an array `a`, this is analogous to setting `a[index]` to the value `newElement`.

Throws:

`ArrayIndexOutOfBoundsException` if the `index` is not greater than or equal to `0` and strictly less than the current size of the vector.

Methods to Add Elements

```
public void add(int index, Object newElement)
```

Inserts `newElement` as an element in the calling vector at the specified index. Each element in the vector with an index greater or equal to `index` is shifted upward to have an index that is `1` greater than the value it had previously.

Note that you can use this method to add an element after the last current element. The capacity of the vector is increased if that is required.

Throws:

`ArrayIndexOutOfBoundsException` if the index is not greater than or equal to `0` and less than *or equal to* the current size of the vector.

```
public boolean add(Object newElement)
```

Adds `newElement` to the end of the calling vector and increases its size by 1. The capacity of the vector is increased if that is required. Returns `true` if successful. Normally used as a `void` method.

```
public void clear( )
```

Removes all elements from the calling vector and sets its size to zero.

Methods to Remove Elements

```
public Object remove(int index)
```

Deletes the element at the specified index and returns the element deleted. Each element in the vector with an index greater than or equal to `index` is decreased to have an index that is 1 less than the value it had previously.

Throws:

`ArrayIndexOutOfBoundsException` if the `index` is not greater than or equal to `0` and less than the current size of the vector.

```
public boolean remove(Object theElement)
```

Removes the first occurrence of `theElement` from the calling vector. If `theElement` is found in the vector, then each element in the vector with an index greater than or equal to `theElement`'s index is decreased to have an index that is 1 less than the value it had previously. Returns `true` if `theElement` was found (and removed). Returns `false` if `theElement` was not found in the calling vector.

Search Methods

```
public boolean contains(Object target)
```

Returns `true` if `target` is an element of the calling vector; otherwise returns `false`.

```
public int indexOf(Object target)
```

Returns the index of the first element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns `–1` if `target` is not found.

```
public int indexOf(Object target, int startIndex)
```

Returns the index of the first element that is equal to `target`, but only considers indices that are greater than or equal to `startIndex`. Uses the method `equals` of the object `target` to test for equality. Returns `–1` if `target` is not found.

```
public boolean isEmpty( )
```

Returns `true` if the calling vector is empty (that is, has size `0`); otherwise returns `false`.

```
public int lastIndexOf(Object target)
```

Returns the index of the last element that is equal to `target`. Uses the method `equals` of the object `target` to test for equality. Returns `–1` if `target` is not found.

```
public Object firstElement( )
```

Returns the first element of the calling vector.

Throws:

`NoSuchElementException` if the vector is empty.

```
public Object lastElement( )
```

Returns the last element of the calling vector.

Throws:

`NoSuchElementException` if the vector is empty.

Iterators

```
public Iterator iterator()
```

Returns an iterator for the calling vector. (Iterators are discussed in Section 14.3.)

```
ListIterator listIterator( )
```

Returns a list iterator for the calling vector. (Iterators are discussed in Section 14.3.)

```
ListIterator listIterator(int index)
```

Returns a list iterator for the calling vector starting at `index`. The first element to be returned by the iterator is the one at `index`. (Iterators are discussed in Section 14.3.)

Converting to an Array

```
public Object[] toArray()
```

Returns an array containing all of the elements in the vector. The elements of the array are indexed the same as in the vector.

```
public Object[] toArray(Object[] a)
```

Returns an array containing all of the elements in the vector. The elements of the array are indexed the same as in the vector.

The argument `a` is used primarily to specify the type of the array returned. The exact details are as follows:

The type of the returned array is that of `a`. If the collection fits in the array `a`, then `a` is used to hold the elements of the returned array; otherwise a new array is created with the same type as `a`.

If `a` has more elements than the vector, the element in `a` immediately following the end of the vector elements is set to `null`.

Throws:

`ArrayStoreException` if the base type of `a` is not an ancestor class of all the elements in the vector.

`NullPointerException` if `a` is `null`.

Memory Management

```
public int capacity( )
```

Returns the current capacity of the calling vector.

```
public void ensureCapacity(int newCapacity)
```

Increases the capacity of the calling vector to ensure that it can hold at least `newCapacity` elements. Using `ensureCapacity` can sometimes increase efficiency, but its use is not needed for any other reason.

```
public void setSize(int newSize)
```

Sets the size of the calling vector to `newSize`. If `newSize` is greater than the current size, the new elements receive the value `null`. If `newSize` is less than the current size, all elements at index `newSize` and greater are discarded.

Throws:

`ArrayIndexOutOfBoundsException` if `newSize` is negative.

```
public int size( )
```

Returns the number of elements in the calling vector.

```
public void trimToSize( )
```

Trims the capacity of the calling vector to be the vector's current size. This is used to save storage.

Make a Copy

```
public Object clone( )
```

Returns a clone of the calling vector. The clone is an identical copy of the calling vector.

Older Methods

These are methods that are not part of the newer collection framework, but are retained for backward compatibility. You should use the above newer methods instead. But, you may find these used in older code.

```
public void addElement(Object newElement)
```

Same as `add`.

```
public void insertElementAt(Object newElement, int index)
```

Same as `add`.

```
public Object elementAt(int index)
```

Same as `get`.

```
public void removeAllElements( )
```

Same as `clear`.

```
public boolean removeElement(Object theElement)
```

Same as `remove`.

```
public void removeElementAt(int index)
```

Same as `remove` but does not return the element removed.

```
public void setElementAt(Object newElement, int index)
```

Same as `set` with the arguments reversed but does not return the element replaced.

`WindowListener` INTERFACE

Package: `java.awt.event`

The `WindowEvent` class is also in this package.

Extends the `EventListener` interface.

```
public void windowActivated(WindowEvent e)
```

Invoked when a window is activated. When you click in a window, it becomes the activated window. Other actions can also activate a window.

```
public void windowClosed(WindowEvent e)
```

Invoked when a window has been closed.

```
public void windowClosing(WindowEvent e)
```

Invoked when a window is in the process of being closed. Clicking the close-window button causes an invocation of this method.

```
public void windowDeactivated(WindowEvent e)
```

Invoked when a window is deactivated. When a window is activated, all other windows are deactivated. Other actions can also deactivate a window.

```
public void windowDeiconified(WindowEvent e)
```

Invoked when a window is deiconified. When you activate a minimized window, it is deiconified.

```
public void windowIconified(WindowEvent e)
```

Invoked when a window is iconified. When you click the minimize button in a `JFrame`, it is iconified.

```
public void windowOpened(WindowEvent e)
```

Invoked when a window has been opened.