Class, Object and Inheritance in Java

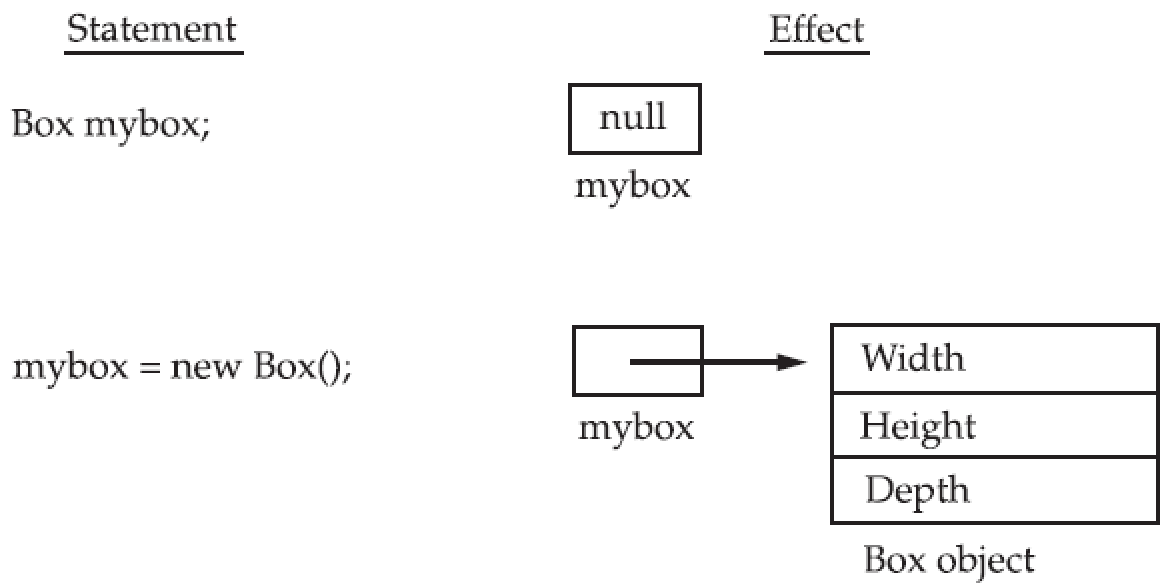
The General Form of a Class:

class *classname {*
type instance-variable1;
type instance-variable2;
// ...
type instance-variableN;

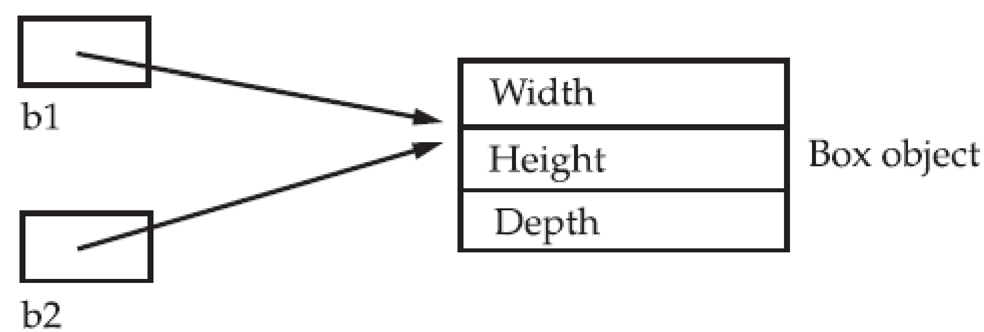
type methodname1(parameter-list) {
// body of method
}
type methodname2(parameter-list) {
// body of method
}
// ...
type methodnameN(parameter-list) {
// body of method
}
}

```
/* A program that uses the Box class.
Call this file BoxDemo.java
*/
class Box {
double width;
double height;
double depth;
}
// This class declares an object of type Box.
class BoxDemo {
public static void main(String args[]) {
Box mybox = new Box();
double vol;
// assign values to mybox's instance variables
mybox.width = 10;
mybox.height = 20;
mybox.depth = 15;
// compute volume of box
vol = mybox.width * mybox.height * mybox.depth;
System.out.println("Volume is " + vol);
}
}
```

- new operator dynamically allocates memory for an object.



Box b1 = new Box();
Box b2 = b1;



StringBuffer and **StringBuilder**

- modifiable string
- defined in **java.lang**
- declared **final** classes
- implement the **CharSequence** interface
- provides much of the functionality of strings

StringBuffer

- **StringBuffer Constructors**

- **StringBuffer defines these four constructors:**
  - StringBuffer( )
  - StringBuffer(int *size)*
  - StringBuffer(String *str)*
  - StringBuffer(CharSequence *chars)*

✓ reserves room for 16 characters without reallocation.

StringBuffer methods

- **length( ) and capacity()**
  - current length and total allocated capacity
  - general forms:

    int length( )

    int capacity( )

```
// StringBuffer length vs. capacity.
class StringBufferDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer = " + sb);
System.out.println("length = " + sb.length());
System.out.println("capacity = " + sb.capacity());
}
}
```

```
buffer = Hello
length = 5
capacity = 21
```

StringBuffer methods

- **ensureCapacity( )**
  - Preallocate
  - general form:

    void ensureCapacity(int *capacity)*

- ✓ Here, *capacity specifies the size of the buffer.*

- **setLength( )**
  - set the length of the buffer within a **StringBuffer object**
  - void setLength(int *len)*

- ✓ *len specifies the length of the buffer. This value must be nonnegative.*

StringBuffer methods

- Increase in the size of the buffer adds null characters to the end
- setLength( ) with a value less than the current value then

  then the characters stored beyond the new length will be lost.

StringBuffer methods

charAt( ) and setCharAt( )

char charAt(int *where)*
void setCharAt(int *where, char ch)*

```
class setCharAtDemo {
public static void main(String args[]) {
StringBuffer sb = new StringBuffer("Hello");
System.out.println("buffer before = " + sb);
System.out.println("charAt(1) before = " + sb.charAt(1));
sb.setCharAt(1, 'i');
sb.setLength(2);
System.out.println("buffer after = " + sb);
System.out.println("charAt(1) after = " + sb.charAt(1));
}
}
```


buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i

StringBuffer methods

✓ A substring of a **StringBuffer** into an **array,** use the **getChars( )** method**.**

void getChars(int *sourceStart, int sourceEnd, char target[ ],*int *targetStart)*

✓**append( )**

- The **append( ) method concatenates.**
- It has several overloaded versions. Here are a few of its forms:

StringBuffer append(String *str)*

StringBuffer append(int *num)*

StringBuffer append(Object *obj)*

StringBuffer methods

```
class appendDemo {
public static void main(String args[]) {
    String s;
    int a = 42;
    StringBuffer sb = new StringBuffer(40);
    s = sb.append("a = ").append(a).append("!").toString();
    System.out.println(s);
    }
}
```

a = 42!

StringBuffer methods

✓ **insert( )**

- The **insert( ) method inserts one string into another.**
- It is overloaded to accept values of all the simple types, plus Strings, Objects, and CharSequences.
- These are a few of its forms:

  StringBuffer insert(int *index, String str)*

  StringBuffer insert(int *index, char ch)*

  StringBuffer insert(int *index, Object obj)*

Here, *index specifies the index at which point the string will be inserted .*

The following sample program inserts “like” between “I” and “Java”:

```
// Demonstrate insert().
class insertDemo {
public static void main(String args[]) {
	StringBuffer sb = new StringBuffer("I Java!");
	sb.insert(2, "like ");
	System.out.println(sb);
	}
}
```

The output of this example is shown here:
I like Java!

✓ reverse( )

You can reverse the characters within a **StringBuffer object using reverse( ),**

```
// Using reverse() to reverse a StringBuffer.
class ReverseDemo {
    public static void main(String args[]) {
        StringBuffer s = new StringBuffer("abcdef");
        System.out.println(s);
        s.reverse();
        System.out.println(s);
    }
}
```

Here is the output produced by the program:

```
abcdef
fedcba
```

✓ **delete( ) and deleteCharAt( )**

- Delete characters within a **StringBuffer** by using these.
- StringBuffer delete(int *startIndex, int endIndex)*
- StringBuffer deleteCharAt(int *loc)*

```
// Demonstrate delete() and deleteCharAt()
class deleteDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("This is a test.");
        sb.delete(4, 7);
        System.out.println("After delete: " + sb);
        sb.deleteCharAt(0);
        System.out.println("After deleteCharAt: " + sb);
        }
    }
```

The following output is produced:

After delete: This a test.

After deleteCharAt: his a test.

✓ **replace( )**

replace one set of characters with another set inside a **StringBuffer** object**.**

Its signature is shown here:

StringBuffer replace(int *startIndex, int endIndex, String str)*

```
// Demonstrate replace()
class replaceDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("This is a test.");
        sb.replace(5, 7, "was");
        System.out.println("After replace: " + sb);
        }
    }
```

Here is the output:

After replace: This was a test.

✓ **substring( )**

- obtain a portion of a **StringBuffer.**

String substring(int *startIndex)*

String substring(int *startIndex, int endIndex)*

Additional **StringBuffer** Methods

Method	**Description**
StringBuffer appendCodePoint(int *ch*)	Appends a Unicode code point to the end of the invoking object. A reference to the object is returned. Added by J2SE 5.
int codePointAt(int *i*)	Returns the Unicode code point at the location specified by *i*. Added by J2SE 5.
int codePointBefore(int *i*)	Returns the Unicode code point at the location that precedes that specified by *i*. Added by J2SE 5.
int codePointCount(int *start*, int *end*)	Returns the number of code points in the portion of the invoking **String** that are between *start* and *end*–1. Added by J2SE 5.
int indexOf(String *str*)	Searches the invoking **StringBuffer** for the first occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int indexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the first occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*)	Searches the invoking **StringBuffer** for the last occurrence of *str*. Returns the index of the match, or –1 if no match is found.
int lastIndexOf(String *str*, int *startIndex*)	Searches the invoking **StringBuffer** for the last occurrence of *str*, beginning at *startIndex*. Returns the index of the match, or –1 if no match is found.

Method	Description
int offsetByCodePoints(int *start*, int *num*)	Returns the index with the invoking string that is *num* code points beyond the starting index specified by *start*. Added by J2SE 5.
CharSequence subSequence(int *startIndex*, int *stopIndex*)	Returns a substring of the invoking string, beginning at *startIndex* and stopping at *stopIndex*. This method is required by the **CharSequence** interface, which is now implemented by **StringBuffer**.
void trimToSize( )	Reduces the size of the character buffer for the invoking object to exactly fit the current contents. Added by J2SE 5.

```
class IndexOfDemo {
    public static void main(String args[]) {
        StringBuffer sb = new StringBuffer("one two one");
        int i;
        i = sb.indexOf("one");
        System.out.println("First index: " + i);
        i = sb.lastIndexOf("one");
        System.out.println("Last index: " + i);
    }
}
```

The output is shown here:

First index: 0

Last index: 8

StringBuilder class in Java

- Identical to **StringBuffer except for** synchronization, that it is **not thread-safe.**
- Not recommended for **multithreading**.
- **StringBuilder** is **faster** in performance.

NMAMIT MOOLE portal on Inet available now

ftp://172.16.3.238/Books/Computers/Java/

Java Beans

Java Beans

- Component-based **reusability** and **interoperability**.
- A designer able to **select** a component,

  understand its capabilities, and

  incorporate it into an application.
- Programs to be assembled.
- Easy to incorporate functionality into existing code.
- To realize these benefits, **a component architecture** is needed.

✓ **Fortunately, Java Beans provides just such an architecture.**

What Is a Java Bean?

- Software component
- Perform a simple or complex function
- Visible or invisible to a user
- Autonomous or distributed
- Reusable in different environments
- No restriction on its capability

Java Beans

Definition of Java Bean

- Java beans are classes that encapsulate many objects into a single object (**the bean**).
- Main parts of JB are properties, events, and methods.
- They are **serializable**, have a **zero-argument** constructor.
- Allow properties using **getter** and **setter** methods.
- A **reusable** software component in Java that can be **manipulated visually** in an application builder tool.

Java Beans

Advantages of Java Beans

- “write-once, run-anywhere” paradigm.
- The properties, events, and methods of a Bean that are exposed to another application can be controlled.
- Auxiliary software can be provided to help configure a Bean.
- The configuration settings of a Bean can be saved in persistent storage and restored at a later time.
- A Bean may register to receive events from other objects and can generate events that are sent to other objects.

✓Questions: What are the advantages of JB?

How component developer benefit from JB.

Introspection

- What is Introspection?
- Design Patterns for Properties
  - Simple Properties
  - Indexed Properties
- Design Patterns for Events
- Methods and Design Patterns
  - Using the BeanInfo Interface

Introspection

- Core of Java Beans
- It is a Process of analyzing a Bean to determine its capabilities
- It is an Essential feature of the Java Beans API (design tool)
- Is a way in which developer expose bean's properties, events, and methods.
- No operation without introspection
- Two ways…

1. Simple naming conventions
   - Introspection mechanisms to infer information about a Bean.
2. Additional class that extends the **BeanInfo interface**
   - Supplies this information.

✓ Both approaches are discussed here.

Design Patterns for Properties

- A ***property is a subset of a Bean's state.***
- *The **values** assigned to the properties determine the* behavior & appearance of that component.
- A property is set through a ***setter method.***
- *A* property is obtained by a ***getter method.***
- *There are two types of properties**: simple and indexed.***

Simple Properties

- A simple property has a single value.
- It can be identified by the following design patterns :

public T getN( )

public void setN(T *arg*)

where N is the name of the property and T is its type

- A read/write property has both of these methods to access its values.
- A read-only property has only a get method.
- A write-only property has only a set method.

Here are three read/write simple properties along with their getter & setter methods:

```
private double depth, height, width;
public double getDepth( ) {
return depth;
}
public void setDepth(double d) {
depth = d;
}
public double getHeight( ) {
return height;
}
public void setHeight(double h) {
height = h;
}
public double getWidth( ) {
return width;
}
public void setWidth(double w) {
width = w;
}
```

Indexed Properties

- An indexed property consists of multiple values.
- It can be identified by the following design patterns:

public T getN(int *index);*

public void setN(int *index, T value);*

public T[ ] getN( );

public void setN(T *values[ ]);*

where N is the name of the property and T is its type

Here is an **indexed property** called **data** along with its **getter and setter** methods**:**

```
private double data[ ];
public double getData(int index) {
return data[index];
}
public void setData(int index, double value) {
data[index] = value;
}

public double[ ] getData( ) {
return data;
}
public void setData(double[ ] values) {
data = new double[values.length];
System.arraycopy(values, 0, data, 0, values.length);
}
```

Design Patterns for Events

- Beans use the delegation event model.
- Beans can generate events and send them to other objects.
- These can be identified by the following design patterns:

public void addTListener(TListener *eventListener)*

public void addTListener(TListener *eventListener)*

throws java.util.TooManyListenersException

public void removeTListener(TListener *eventListener)*

where **T is the type of the event**

✓Questions: How do you multicast and unicast a event in Java beans?

Illustrate with code scripts multicast and unicast events in Java beans.

For example, assuming an event interface type called **TemperatureListener, a Bean that monitors** temperature might supply the following methods:

```
public void addTemperatureListener(TemperatureListener tl) {
...
}
public void removeTemperatureListener(TemperatureListener tl) {
...
}
```

Methods and Design Patterns

Using the BeanInfo Interface

- This **interface enables you to *explicitly control what*** information is available.
- The **BeanInfo interface defines several methods, including these:**
    - PropertyDescriptor[ ] getPropertyDescriptors( )
    - EventSetDescriptor[ ] getEventSetDescriptors( )
    - MethodDescriptor[ ] getMethodDescriptors( )

They return arrays of objects provide information about the properties, events, and methods of a Bean.
The classes **PropertyDescriptor, EventSetDescriptor, and MethodDescriptor** are defined within the **java.beans package.**

Using the BeanInfo Interface

- When creating a class that implements **BeanInfo,** you must call that class ***bnameBeanInfo,*** where *bname is the name of the Bean.*
- *For example, if the Bean is called* ***MyBean***, then the information class must be called **MyBeanBeanInfo.**

SimpleBeanInfo class of Java Beans :

- To simplify the use of **BeanInfo, JavaBeans supplies the SimpleBeanInfo class.**
- **It provides** default implementations of the **BeanInfo interface**, including the **three methods** just shown above**.**
- You can **extend** this class and **override** one or more of the methods to explicitly control what aspects of a Bean are exposed.
- If you don't override a method, then design-pattern introspection will be used. For example, if you don't override **getPropertyDescriptors( ),** then design patterns are used to discover a Bean's properties.
- You will see **SimpleBeanInfo** in action in later discussion.

The Java Beans API

The Java Beans functionality is provided by a set of classes and interfaces in the **java.beans** package.

4 are of particular interest:

Introspector,

PropertyDescriptor,

EventSetDescriptor, and

MethodDescriptor.

The Interfaces in **java.beans**

AppletInitializer	Methods in this interface are used to initialize Beans that are also applets.
BeanInfo	This interface allows a designer to specify information about the properties, events, and methods of a Bean.
Customizer	This interface allows a designer to provide a graphical user interface through which a Bean may be configured.
DesignMode	Methods in this interface determine if a Bean is executing in design mode.
ExceptionListener	A method in this interface is invoked when an exception has occurred
PropertyChangeListener	A method in this interface is invoked when a bound property is changed
PropertyEditor	Objects that implement this interface allow designers to change and display property values.
VetoableChangeListener	A method in this interface is invoked when a constrained property is changed.
Visibility	Methods in this interface allow a Bean to execute in environments where a graphical user interface is not available.

The Classes in **java.beans**

Class	**Description**
BeanDescriptor	This class provides information about a Bean. It also allows you to associate a customizer with a Bean.
Beans	This class is used to obtain information about a Bean.
DefaultPersistenceDelegate	A concrete subclass of **PersistenceDelegate**.
Encoder	Encodes the state of a set of Beans. Can be used to write this information to a stream.
EventHandler	Supports dynamic event listener creation.
EventSetDescriptor	Instances of this class describe an event that can be generated by a Bean.
Expression	Encapsulates a call to a method that returns a result.
FeatureDescriptor	This is the superclass of the **PropertyDescriptor**, **EventSetDescriptor**, and **MethodDescriptor** classes.

The Classes in **java.beans**

IndexedPropertyChangeEvent	A subclass of **PropertyChangeEvent** that represents a change to an indexed property.
IndexedPropertyDescriptor	Instances of this class describe an indexed property of a Bean.
IntrospectionException	An exception of this type is generated if a problem occurs when analyzing a Bean.
Introspector	This class analyzes a Bean and constructs a **BeanInfo** object that describes the component.

Introspector class

- The **Introspector** class provides several static methods that support introspection.
- Important one is **getBeanInfo( ).**
- This method returns a **BeanInfo object,** information about the Bean.
- The **getBeanInfo( ) method** has several forms.
- One shown here:

  static BeanInfo getBeanInfo(Class<?> *bean) throws*

    IntrospectionException

- The returned object contains information about the Bean specified by *bean.*

PropertyDescriptor

The **PropertyDescriptor** class describes a Bean property.

It supports several methods that manage and describe properties.

For example, you can determine if a property is bound by calling isBound( ).

To determine if a property is constrained, call isConstrained( ).

You can obtain the name of property by calling getName( ).

EventSetDescriptor

The **EventSetDescriptor** class represents a Bean **event**.

It supports several methods that obtain the methods that a Bean uses to **add** or **remove event listeners**, and to otherwise **manage** events.

For example, to obtain the method used to **add listeners**, call

getAddListenerMethod( ).

To obtain the method used **to remove listeners**, call

getRemoveListenerMethod( ).

To obtain the type of a **listener**, call

getListenerType( ).

You can obtain the name of an **event** by calling

getName( ).

MethodDescriptor

The **MethodDescriptor** class represents a Bean method.

To obtain the name of the method, call

getName( ).

You can obtain information about the method by calling

Method **getMethod( ),**

An object of type **Method** that describes the method is returned.

A Java Bean Program Example

The example uses three classes.
The first is a Bean called **Colors,** shown here:

```
// A simple Bean.
import java.awt.*;
import java.awt.event.*;
import java.io.Serializable;

public class Colors extends Canvas implements Serializable {
    transient private Color color; // not persistent
    private boolean rectangular; // is persistent

    public Colors() {
    addMouseListener(new MouseAdapter() {
    public void mousePressed(MouseEvent me) {
    change();
    }
    });
```

```
rectangular = false;
setSize(200, 100);
change();
} // End of Color constructor

public boolean getRectangular() {
return rectangular;
}

public void setRectangular(boolean flag) {
this.rectangular = flag;
repaint();
}

public void change() {
color = randomColor();
repaint();
}
```

```
private Color randomColor() {
int r = (int)(255*Math.random());
int g = (int)(255*Math.random());
int b = (int)(255*Math.random());
return new Color(r, g, b);
}

public void paint(Graphics g) {
	Dimension d = getSize();
	int h = d.height;
	int w = d.width;
	g.setColor(color);
	if(rectangular) {
		g.fillRect(0, 0, w-1, h-1);
	}
	else {
		g.fillOval(0, 0, w-1, h-1);
	}
} //End of paint method
} // End of Color class
```

- The **Colors Bean** displays a colored object within a frame**.**
- **The color of the component is** determined by the private Color variable **color.**
- Its shape is determined by the private **boolean** variable **rectangular.**
- The constructor defines **MouseAdapter** and **overrides** its **mousePressed( )** method**.** The **change( )** method is invoked in response to mouse presses.
- It selects a random color and then repaints the component.
- The **getRectangular( ) and setRectangular( )** methods provide access to the one property of this Bean.
- The **change( )** method calls **randomColor( )** to choose a color & then calls **repaint( )** to make the change visible.

Here, the first argument of PropertyDescriptor constructor is the name of the property, and the second argument is the class of the Bean.

```
// A Bean information class.
import java.beans.*;
public class ColorsBeanInfo extends SimpleBeanInfo {
    public PropertyDescriptor[] getPropertyDescriptors() {
    try {
        PropertyDescriptor rectangular = new
                    PropertyDescriptor("rectangular", Colors.class);
        PropertyDescriptor pd[] = {rectangular};
        return pd;
      }
    catch(Exception e) {
        System.out.println("Exception caught. " + e);
      }
    return null;
    } // End of getPropertyDescriptors method
} //End of ColorsBeanInfo class
```

The final 3rd class is called **IntrospectorDemo.** It uses introspection to display the properties and events that are available within the Colors Bean.

```
// Show properties and events.
import java.awt.*;
import java.beans.*;
public class IntrospectorDemo {
    public static void main(String args[]) {
    try {
        Class c = Class.forName("Colors");
        BeanInfo beanInfo = Introspector.getBeanInfo(c);
        System.out.println("Properties:");
        PropertyDescriptor propertyDescriptor[] =
            beanInfo.getPropertyDescriptors();
        for(int i = 0; i < propertyDescriptor.length; i++) {
            System.out.println("\t" + propertyDescriptor[i].getName());
        }
```

```
System.out.println("Events:");
EventSetDescriptor eventSetDescriptor[] =
beanInfo.getEventSetDescriptors();
for(int i = 0; i < eventSetDescriptor.length; i++) {
    System.out.println("\t" + eventSetDescriptor[i].getName());
} //End of for loop
} //End of try block
catch(Exception e) {
System.out.println("Exception caught. " + e);
} //End of catch
} // End of main method
} // End of IntrospectorDemo  class
```

The output from this program is the following:

Properties:

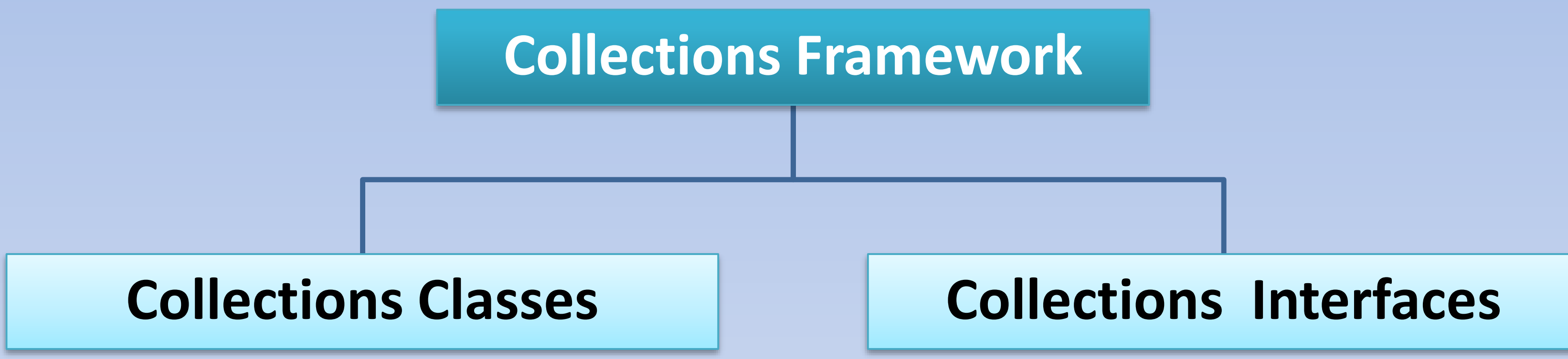
- rectangular

Events:

- mouseWheel
- mouse
- mouseMotion
- component
- hierarchyBounds
- focus
- hierarchy
- propertyChange
- inputMethod
- key

- Notice two things in the output.
- First, because **ColorsBeanInfo** overrides **getPropertyDescriptors( ) such that the only property** returned is **rectangular**, only the **rectangular** property is displayed.
- However**,** because **getEventSetDescriptors( )** is not overridden by **ColorsBeanInfo, design-pattern introspection** is used, and all events are found, including those in **Colors' superclass, Canvas.**
- **Remember,** if you don't override one of the "get" methods defined by **SimpleBeanInfo,** then the default, design-pattern introspection is used.
- To observe the difference that **ColorsBeanInfo** makes, erase **its class file** and then run **IntrospectorDemo** again**. This time it will report more properties.**

Collections Classes and Collections Interfaces


Collections Framework
Collections Classes
Collections Interfaces

- Groups of objects.
- Collections Framework standardizes the way in which groups of objects are handled by your programs.
- Collections Framework was designed to meet several goals.
  1. High-performance.

     The implementations for the fundamental collections are highly efficient.
  2. Uniformity in work and high degree of interoperability .

     Allows different types of collections to work in a similar manner.
  3. Extending and/or adapting a collection is easy.

     Collections Framework is built upon a set of standard interfaces.
  4. Mechanisms to integrate the standard arrays into the Collections Framework.
- ***Algorithms*** as static methods within the **Collections class.**
- ***Iterator*** a general-purpose, standardized way of accessing the elements within a collection, one at a time.

- Framework defines several **map** interfaces and classes

  -store **key/value** pairs

  -contents of a map as a collection

- **java.util** package contains this most powerful subsystems of Java :

  The ***Collections Framework.***

A collections framework is a unified architecture for representing and manipulating collections. All collections frameworks contain the following −

Interfaces

− These are abstract data types that represent collections.

− In object-oriented languages, interfaces generally form a hierarchy.

Implementations, i.e., Classes

− These are the concrete implementations of the collection interfaces.

− They are reusable data structures.

Algorithms

− These are the methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.

− The algorithms are said to be polymorphic: that is, the same method can be used on many different implementations of the appropriate collection interface.

Classes of java.util :

AbstractCollection	EventObject	Random
AbstractList	FormattableFlags	ResourceBundle
AbstractMap	Formatter	Scanner
AbstractQueue	GregorianCalendar	ServiceLoader (Added by Java SE 6.)
AbstractSequentialList	HashMap	SimpleTimeZone
AbstractSet	HashSet	Stack
ArrayDeque (Added by Java SE 6.)	Hashtable	StringTokenizer
ArrayList	IdentityHashMap	Timer
Arrays	LinkedHashMap	TimerTask
BitSet	LinkedHashSet	TimeZone
Calendar	LinkedList	TreeMap
Collections	ListResourceBundle	TreeSet
Currency	Locale	UUID
Date	Observable	Vector
Dictionary	PriorityQueue	WeakHashMap
EnumMap	Properties	
EnumSet	PropertyPermission	
EventListenerProxy	PropertyResourceBundle	

Interfaces of java.util :

Collection	List	Queue
Comparator	ListIterator	RandomAccess
Deque (Added by Java SE 6.)	Map	Set
Enumeration	Map.Entry	SortedMap
EventListener	NavigableMap (Added by Java SE 6.)	SortedSet
Formattable	NavigableSet (Added by Java SE 6.)	
Iterator	Observer	

The **Interfaces** that we are going to study:

1. **Collection**
2. **List**
3. **Set**
4. **SortedSet**
5. **NavigableSet**
6. **Queue**
7. **Deque**

The Collection Interfaces

- The Collections Framework defines several interfaces.
- They determine the fundamental nature of the collection classes.
- The interfaces that underpin collections are summarized in the following table:

Interface	Description
Collection	Enables you to work with groups of objects; it is at the top of the collections hierarchy.
Deque	Extends **Queue** to handle a double-ended queue. (Added by Java SE 6.)
List	Extends **Collection** to handle sequences (lists of objects).
NavigableSet	Extends **SortedSet** to handle retrieval of elements based on closest-match searches. (Added by Java SE 6.)
Queue	Extends **Collection** to handle special types of lists in which elements are removed only from the head.
Set	Extends **Collection** to handle sets, which must contain unique elements.
SortedSet	Extends **Set** to handle sorted sets.

- Collections also use **Comparator, Iterator,** & **ListIterator** and **RandomAccess** interfaces**.**
- **Comparator** defines how two objects are compared.
- **Iterator** and **ListIterator** enumerate the objects within a collection.
- **RandomAccess** supports efficient, random access to its elements.

The **Collection** interface**:**

- It is the foundation upon which the Collections Framework is built.
- **Collection is a generic** interface that has this declaration:

  interface Collection<E>

  E specifies the type of objects that the collection will hold.
- Collection extends the **Iterable** interface.

  - all collections can be cycled through by use of the **for-each** style for loop.
- Collection declares the core methods.

  - add(), addAll(), remove(), removeAll() etc. explained later.

- Several of methods can throw an **UnsupportedOperationException.**
- A **ClassCastException** is generated when one object is incompatible with Another.
- A **NullPointerException** is thrown if an attempt is made to store a null object and null elements are not allowed in the collection.
- An **IllegalArgumentException** is thrown if an invalid argument is used.
- An **IllegalStateException** is thrown if an attempt is made to add an element to a fixed-length collection that is full.

- Objects (of type **E**) are added to a collection by calling **add( ).**
- Add the entire contents of one collection to another by calling **addAll( ).**
- You can remove an object by using **remove( ).**
- To remove a group of objects**, call removeAll( ).**
- All elements cab be removed except for of a specified group by calling **retainAll**( ).
- To empty a collection, call **clear( ).**

contains( ) - determine whether a collection contains a specific object

containsAll( ) - whether one collection contains all the members of another

isEmpty( ) - determine whether a collection is empty or not

size( ) - The number of elements currently held in a collection

The **toArray( )** methods

- return an array that contains the elements in the invoking collection. (array of Object**,** same type as the array specified)

equals( ) - two collections can be compared for equality

iterator( ) - returns an iterator to a collection

Method	Description
boolean add(E *obj*)	Adds *obj* to the invoking collection. Returns **true** if *obj* was added to the collection. Returns **false** if *obj* is already a member of the collection and the collection does not allow duplicates.
boolean addAll(Collection<? extends E> *c*)	Adds all the elements of *c* to the invoking collection. Returns **true** if the operation succeeded (i.e., the elements were added). Otherwise, returns **false**.
void clear( )	Removes all elements from the invoking collection.
boolean contains(Object *obj*)	Returns **true** if *obj* is an element of the invoking collection. Otherwise, returns **false**.
boolean containsAll(Collection<?> *c*)	Returns **true** if the invoking collection contains all elements of *c*. Otherwise, returns **false**.
boolean equals(Object *obj*)	Returns **true** if the invoking collection and *obj* are equal. Otherwise, returns **false**.
int hashCode( )	Returns the hash code for the invoking collection.
boolean isEmpty( )	Returns **true** if the invoking collection is empty. Otherwise, returns **false**.
Iterator<E> iterator( )	Returns an iterator for the invoking collection.
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.

Iterator<E> iterator( )	Returns an iterator for the invoking collection.
boolean remove(Object *obj*)	Removes one instance of *obj* from the invoking collection. Returns **true** if the element was removed. Otherwise, returns **false**.
boolean removeAll(Collection<?> *c*)	Removes all elements of *c* from the invoking collection. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
boolean retainAll(Collection<?> *c*)	Removes all elements from the invoking collection except those in *c*. Returns **true** if the collection changed (i.e., elements were removed). Otherwise, returns **false**.
int size( )	Returns the number of elements held in the invoking collection.
Object[ ] toArray( )	Returns an array that contains all the elements stored in the invoking collection. The array elements are copies of the collection elements.
<T> T[ ] toArray(T *array*[ ])	Returns an array that contains the elements of the invoking collection. The array elements are copies of the collection elements. If the size of *array* equals the number of elements, these are returned in *array*. If the size of *array* is less than the number of elements, a new array of the necessary size is allocated and returned. If the size of *array* is greater than the number of elements, the array element following the last collection element is set to **null**. An **ArrayStoreException** is thrown if any collection element has a type that is not a subtype of *array*.

```
…..
ArrayList al = new ArrayList();
…..
// add elements to the array list
al.add("C");
al.add("A");
al.add("E");
al.add("B");
al.add("D");
al.add("F");
…..
// Use iterator to display contents of al
Iterator itr = al.iterator();
…..
while(itr.hasNext()) {
    data_type element = itr.next();
……..
}
```

For example, the class **Employee** might implement its hash function by composing the hashes of its members:

```java
public class Employee {
    int        employeeId;
    String     name;
    Department dept;

    // other methods would be in here

    @Override
    public int hashCode() {
        int hash = 1;
        hash = hash * 17 + employeeId;
        hash = hash * 31 + name.hashCode();
        hash = hash * 13 + (dept == null ? 0 : dept.hashCode());
        return hash;
    }
}
```

The **List** Interface

- The List interface **extends Collection** and declares the behavior of a collection.
- Stores a sequence of elements.
- Elements can be inserted or accessed by their position in the list using a **zero-based index.**
- **List is a generic** interface that has this declaration:

  interface List<E>

  Here, E specifies the type of objects that the list will hold.

- In addition to the methods defined by **Collection, List** defines some of its own, which are summarized in below.

	returns **false** otherwise.
E get(int *index*)	Returns the object stored at the specified index within the invoking collection.
int indexOf(Object *obj*)	Returns the index of the first instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
int lastIndexOf(Object *obj*)	Returns the index of the last instance of *obj* in the invoking list. If *obj* is not an element of the list, –1 is returned.
ListIterator<E> listIterator( )	Returns an iterator to the start of the invoking list.
ListIterator<E> listIterator(int *index*)	Returns an iterator to the invoking list that begins at the specified index.
E remove(int *index*)	Removes the element at position *index* from the invoking list and returns the deleted element. The resulting list is compacted. That is, the indexes of subsequent elements are decremented by one.
E set(int *index*, E *obj*)	Assigns *obj* to the location specified by *index* within the invoking list.
List<E> subList(int *start*, int *end*)	Returns a list that includes elements from *start* to *end*–1 in the invoking list. Elements in the returned list are also referenced by the invoking object.

- **UnsupportedOperationException** if the list cannot be modified
- **ClassCastException**
- **IndexOutOfBoundsException**
- **NullPointerException** is thrown if an attempt is made to store a null object and null elements are not allowed in the list.
- An **IllegalArgumentException** is thrown if an invalid argument is used.

The **Set** Interface

- The **Set interface** defines a **set.**
- It extends Collection and declares the behavior of a collection that does not allow duplicate elements.
- Therefore, the **add( )** method returns **false** if an attempt is made to add duplicate elements to a set.
- It does not define any additional methods of its own.
- **Set is a generic interface that has this declaration:**
- interface Set<E>
- Here, E specifies the type of objects that the set will hold.

The **SortedSet** Interface

- The **SortedSet** interface extends **Set** and declares the behavior of a set **sorted in ascending order**.
- **SortedSet** is a generic interface that has this declaration:

  interface SortedSet<E>

  Here, E specifies the type of objects that the set will hold.
- Addition to those methods defined by **Set, the SortedSet interface declares the methods** summarized as follows.

Method	Description
E first( )	Returns the first element in the invoking sorted set.
SortedSet<E> headSet(E *end*)	Returns a **SortedSet** containing those elements less than *end* that are contained in the invoking sorted set. Elements in the returned sorted set are also referenced by the invoking sorted set.
E last( )	Returns the last element in the invoking sorted set.
SortedSet<E> subSet(E *start*, E *end*)	Returns a **SortedSet** that includes those elements between *start* and *end*–1. Elements in the returned collection are also referenced by the invoking object.
SortedSet<E> tailSet(E *start*)	Returns a **SortedSet** that contains those elements greater than or equal to *start* that are contained in the sorted set. Elements in the returned set are also referenced by the invoking object.

```java
    tree.add(12);
    tree.add(13);
    tree.add(14);
    tree.add(15);
    tree.add(16);
    tree.add(17);

    // getting values less than 15
    treeheadset = (TreeSet)tree.headSet(15);

    // creating iterator
    Iterator iterator;
    iterator = treeheadset.iterator();

    //Displaying the tree set data
    System.out.println("Tree set data: ");

    while (iterator.hasNext()) {
       System.out.println(iterator.next() + " ");
    }
  }
```

```
Tree set data:
12
13
14
```

The **NavigableSet** Interface

The **NavigableSet** interface was added by **Java SE 6.** (SE – Standard Edition)

It extends **SortedSet** and declares the behavior of a collection that supports the retrieval of elements based on the **closest match to a given value or values**.

NavigableSet is a generic interface that has this declaration:

interface NavigableSet<E>

Here, E specifies the type of objects that the set will hold.

In addition to the methods that it inherits from SortedSet, NavigableSet adds those summarized in Table below:

Method	Description
E ceiling(E *obj*)	Searches the set for the smallest element *e* such that *e* >= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the greatest to least. In other words, it returns a reverse iterator.
NavigableSet<E> descendingSet( )	Returns a **NavigableSet** that is the reverse of the invoking set. The resulting set is backed by the invoking set.
E floor(E *obj*)	Searches the set for the largest element *e* such that *e* <= *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
NavigableSet<E> headSet(E *upperBound*, boolean *incl*)	Returns a **NavigableSet** that includes all elements from the invoking set that are less than *upperBound*. If *incl* is **true**, then an element equal to *upperBound* is included.
E higher(E *obj*)	Searches the set for the largest element *e* such that *e* > *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.
E lower(E *obj*)	Searches the set for the largest element *e* such that *e* < *obj*. If such an element is found, it is returned. Otherwise, **null** is returned.

The **Queue** Interface

- The **Queue** interface extends Collection and declares the behavior of a queue, which is often a **first-in, first-out list**.
- However, there are types of queues in which the ordering is based upon other criteria.
- **Queue** is a generic interface that has this declaration:

interface Queue<E>

Method	Description
E element( )	Returns the element at the head of the queue. The element is not removed. It throws **NoSuchElementException** if the queue is empty.
boolean offer(E *obj*)	Attempts to add *obj* to the queue. Returns **true** if *obj* was added and **false** otherwise.
E peek( )	Returns the element at the head of the queue. It returns **null** if the queue is empty. The element is not removed.
E poll( )	Returns the element at the head of the queue, removing the element in the process. It returns **null** if the queue is empty.
E remove( )	Removes the element at the head of the queue, returning the element in the process. It throws **NoSuchElementException** if the queue is empty.

The **Deque** Interface

- The **Deque** interface was added by Java SE 6.
- It extends **Queue** and declares the behavior of a double-ended queue.
- Double-ended queues can function as standard, first-in, first-out queues or as last-in, first-out stacks.
- Deque is a generic interface that has this declaration: **interface Deque<E>**

Method	Description
void addFirst(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
void addLast(E *obj*)	Adds *obj* to the tail of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.
Iterator<E> descendingIterator( )	Returns an iterator that moves from the tail to the head of the deque. In other words, it returns a reverse iterator.
E getFirst( )	Returns the first element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.
E getLast( )	Returns the last element in the deque. The object is not removed from the deque. It throws **NoSuchElementException** if the deque is empty.

boolean offerFirst(E *obj*)	Attempts to add *obj* to the head of the deque. Returns **true** if *obj* was added and **false** otherwise. Therefore, this method returns **false** when an attempt is made to add *obj* to a full, capacity-restricted deque.
boolean offerLast(E *obj*)	Attempts to add *obj* to the tail of the deque. Returns **true** if *obj* was added and **false** otherwise.
E peekFirst( )	Returns the element at the head of the deque. It returns **null** if the deque is empty. The object is not removed.
E peekLast( )	Returns the element at the tail of the deque. It returns **null** if the deque is empty. The object is not removed.
E pollFirst( )	Returns the element at the head of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pollLast( )	Returns the element at the tail of the deque, removing the element in the process. It returns **null** if the deque is empty.
E pop( )	Returns the element at the head of the deque, removing it in the process. It throws **NoSuchElementException** if the deque is empty.
void push(E *obj*)	Adds *obj* to the head of the deque. Throws an **IllegalStateException** if a capacity-restricted deque is out of space.

E removeFirst( )	Returns the element at the head of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeFirstOccurrence(Object *obj*)	Removes the first occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.
E removeLast( )	Returns the element at the tail of the deque, removing the element in the process. It throws **NoSuchElementException** if the deque is empty.
boolean removeLastOccurrence(Object *obj*)	Removes the last occurrence of *obj* from the deque. Returns **true** if successful and **false** if the deque did not contain *obj*.

- Now that you are familiar with the collection interfaces.
- You are ready to examine the standard classes that implement them.

-: Collection Classes:-

- Some of the classes are full implementations that can be used as-is.
- Others are abstract, providing skeletal implementations that are used as starting points for creating concrete collections.

AbstractCollection	- Implements most of the Collection interface.
AbstractList	- Extends AbstractCollection and implements most of the List interface.
AbstractQueue	- Extends AbstractCollection and implements parts of the Queue interface.
AbstractSequentialList	- Extends AbstractList for use by a collection that uses sequential rather than random access of its elements.
✓**LinkedList**	- Implements a linked list by extending AbstractSequentialList.
✓**ArrayList**	- Implements a dynamic array by extending AbstractList.

✓**ArrayDeque**	Implements a dynamic double-ended queue by extending AbstractCollection and implementing the Deque interface.
AbstractSet	Extends AbstractCollection and implements most of the Set interface.
✓**EnumSet**	Extends AbstractSet for use with enum elements.
✓**HashSet**	Extends AbstractSet for use with a hash table.
✓**LinkedHashSet**	Extends HashSet to allow insertion-order iterations.
✓**PriorityQueue**	Extends AbstractQueue to support a priority-based queue.
✓**TreeSet**	Implements a set stored in a tree. Extends AbstractSet.

ArrayList Class

- The **ArrayList** class extends **AbstractList** and implements the **List interface.**
- **ArrayList is a** generic class that has this declaration:

  class ArrayList<E>

  Here, **E specifies the type of objects that the list will hold.**
- **ArrayList** is a **variable-length** array of object references.
- That is, an **ArrayList** can **dynamically increase** or **decrease in size.**
- They are **created** with an **initial size.**
- When this size is **exceeded**, the collection is **automatically enlarged**.
- When objects are **removed**, the array can be **shrunk**.

ArrayList has the constructors shown here:

ArrayList( )

ArrayList(Collection<? extends E> *c)*

ArrayList(int *capacity)*

- The first constructor builds an empty array list.
- The second constructor builds an array list that is initialized with the elements of the collection *c.*
- The third constructor builds an array list that has the specified initial capacity.
- The capacity grows automatically as elements are added to an array list.

```java
// Demonstrate ArrayList.
import java.util.*;
class ArrayListDemo {
        public static void main(String args[]) {
                // Create an array list.
                ArrayList<String> al = new ArrayList<String>();
                System.out.println("Initial size of al: " +al.size());
                // Add elements to the array list.
                al.add("C");
                al.add("A");
                al.add("E");
                al.add("B");
                al.add("D");
                al.add("F");
                al.add(1, "A2");
                System.out.println("Size of al after additions: " + al.size());
```

```
        System.out.println("Contents of al: " + al);
        // Remove elements from the array list.
        al.remove("F");
        al.remove(2);
        System.out.println("Size of al after deletions: " +al.size());
        System.out.println("Contents of al: " + al);
    } //End of main()
} //End of class
```

The output from this program is shown here:

```
Initial size of al: 0
Size of al after additions: 7
Contents of al: [C, A2, A, E, B, D, F]
Size of al after deletions: 5
Contents of al: [C, A2, E, B, D]
```

ArrayList methods recalled:

- ✓ void **ensureCapacity**(int *cap)*

  – ***increase the*** size of the ArrayList by ***cap.***

- ✓ void **trimToSize( )**

  – **reduce** the size to the elements in ArrayList.

How do you obtaining an Array from an ArrayList explain with illustrations.

Obtaining an Array from an ArrayList:

- ✓ When working with **ArrayList, you will sometimes want to obtain an actual array that contains** the contents of the list.
- ✓ You can do this by calling **toArray( ),** which is defined by **Collection.**
- ✓ Several reasons may exist to convert a collection into an array, such as:
    - To obtain **faster processing** times for certain operations
    - To pass an array to a method that **is not overloaded** to accept a collection
    - To integrate collection-based code with **legacy** code that **does not understand collections**

There are two versions of **toArray( ),**

- **Object[ ]   toArray( )**
- **<T> T[ ]   toArray(T** ***array[ ])***

```java
// Convert an ArrayList into an array.
import java.util.*;
class ArrayListToArray {
		public static void main(String args[]) {
			// Create an array list.
			ArrayList<Integer> al = new ArrayList<Integer>();
			// Add elements to the array list.
			al.add(1);
			al.add(2);
			al.add(3);
			al.add(4);
			System.out.println("Contents of al: " + al);
			// Get the array.
			Integer ia[] = new Integer[al.size()];
			ia = al.toArray(ia);
			int sum = 0;
			// Sum the array.
			for(int i : ia) sum += i;
				System.out.println("Sum is: " + sum);
		}
}
```

The output from the program is shown here:

```
Contents of al: [1, 2, 3, 4]
Sum is: 10
```

LinkedList Class

The **LinkedList** class extends **AbstractSequentialList** and implements the **List, Deque,** and **Queue** interfaces**.**

It provides a **linked-list data structure.**

LinkedList is a generic class that has this declaration:

class LinkedList<E>

It has two constructors:

LinkedList( )

LinkedList(Collection<? extends E> *c)*

- The first constructor builds an empty linked list.
- The second constructor builds a linked list that is initialized with the elements of the collection *c.*

```
// Demonstrate LinkedList.
import java.util.*;
class LinkedListDemo {
        public static void main(String args[]) {
                // Create a linked list.
                LinkedList<String> ll = new LinkedList<String>();
                // Add elements to the linked list.
                ll.add("F");
                ll.add("B");
                ll.add("D");
                ll.add("E");
                ll.add("C");
                ll.addLast("Z");
                ll.addFirst("A");
                ll.add(1, "A2");
                System.out.println("Original contents of ll: " + ll);
                // Remove elements from the linked list.
                ll.remove("F");
                ll.remove(2);
                System.out.println("Contents of ll after deletion: “+ ll);
```

Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]

```
// Remove first and last elements.
ll.removeFirst();
ll.removeLast();
System.out.println("ll after deleting first and last: “+ ll);
// Get and set a value.
String val = ll.get(2);
ll.set(2, val + " Changed");
System.out.println("ll after change: " + ll);
} End of main
}//End of class LinkedListDemo
```

Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]

HashSet Class

HashSet extends **AbstractSet** and implements the **Set** interface**.**

It creates a collection that uses a **hash table for storage.**

HashSet is a generic class that has this declaration:

class HashSet<E>

Here, E specifies the type of objects that the set will hold.

A hash table stores information by using a mechanism called **hashing**.

Hash code is *the informational content of a key is used to determine a unique value.*

The hash code is then used as the index at which the data associated with

the key is stored.

The transformation of the key into its hash code is performed automatically—

you never see the hash code itself.

Also, your code can't directly index the hash table.

The advantage of hashing is that it allows the execution time of **add( ), contains( ),**

remove( ), and size( ) to remain constant even for large sets.

The following constructors are defined:

HashSet( )

HashSet(Collection<? extends E> *c)*

HashSet(int *capacity)*

HashSet(int *capacity, float fillRatio)*

The first form constructs a default hash set.

The second form initializes the hash set by using the elements of *c.*

The third form initializes the capacity of the hash set to capacity. (The default capacity is 16.)

The fourth form initializes both the capacity and the fill ratio (also called *load capacity) of the hash set from its arguments.*

The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward.

For constructors that do not take a fill ratio, 0.75 is used.

HashSet does not define any additional methods beyond those provided by its superclasses and interfaces.

```
// Demonstrate HashSet.
import java.util.*;
class HashSetDemo {
	public static void main(String args[]) {
		// Create a hash set.
		HashSet<String> hs = new HashSet<String>();
		// Add elements to the hash set.
		hs.add("B");
		hs.add("A");
		hs.add("D");
		hs.add("E");
		hs.add("C");
		hs.add("F");
		System.out.println(hs);
	}
}
```

The following is the output from this program:

[D, A, F, C, B, E]

LinkedHashSet Class

The **LinkedHashSet** class extends **HashSet** and adds no members of its own**.**

It is a generic class that has this declaration:

class LinkedHashSet<E>

LinkedHashSet maintains entries in the set, in the order in which they were inserted.

That is, when cycling through a LinkedHashSet using an iterator, the elements will be returned in the order in which they were inserted.

To see the effect of LinkedHashSet, try substituting LinkedHashSet for HashSet in the preceding program.

The output will be

[B, A, D, E, C, F]

which is the order in which the elements were inserted.

TreeSet Class

TreeSet extends **AbstractSet** and implements the **NavigableSet** interface**.**

It creates a collection that uses a tree for storage.

Objects are stored in sorted, ascending order.

Accessand retrieval times are quite fast.

TreeSet is an **excellent choice** when storing large amounts of sorted information that must be found quickly.

TreeSet is a generic class that has this declaration:

Class TreeSet<E>

Here, **E** specifies the type of objects that the set will hold.

TreeSet has the following constructors:

TreeSet( )

TreeSet(Collection<? extends E> *c*)

TreeSet(Comparator<? super E> *comp*)

TreeSet(SortedSet<E> *ss*)

<? super E> means some type which is an ancestor of E.

The first form constructs an empty tree set that will be sorted in ascending order according to the natural order of its elements.

The second form builds a tree set that contains the elements of *c.*

The third form constructs an empty tree set that will be sorted according to the comparator specified by comp.

The fourth form builds a tree set that contains the elements of *ss.*

```
// Demonstrate TreeSet.
import java.util.*;
class TreeSetDemo {
    public static void main(String args[]) {
        // Create a tree set.
        TreeSet<String> ts = new TreeSet<String>();
        // Add elements to the tree set.
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");
        System.out.println(ts);
    }
}
```

The output from this program is shown here:

[A, B, C, D, E, F]

Because **TreeSet** implements the **NavigableSet interface** you can use the methods defined by **NavigableSet** to retrieve elements of a **TreeSet.**

To obtain a subset of **ts** that contains the elements between **C (inclusive) and F (exclusive).** It then displays the resulting set.

```
System.out.println(ts.subSet("C", "F"));
```

The output from this statement is shown here:

[C, D, E]

✓You might want to experiment with the other methods defined by **NavigableSet.**

```
// Use a custom comparator.
import java.util.*;
class MyComp implements Comparator<String> {
        public int compare(String a, String b) {
            String aStr, bStr;
            aStr = a;
            bStr = b;
            return bStr.compareTo(aStr);
        }
}
```

```
class CompDemo {
    public static void main(String args[]) {
        // Create a tree set.
        TreeSet<String> ts = new TreeSet<String>(new MyComp());
        // Add elements to the tree set.
        ts.add("C");
        ts.add("A");
        ts.add("B");
        ts.add("E");
        ts.add("F");
        ts.add("D");
        // Display the elements.
        for(String element : ts)
                System.out.print(element + " ");
        System.out.println();
    }
}
```

As the following output shows, the tree is now stored in reverse order:

F E D C B A

PriorityQueue Class

PriorityQueue extends **AbstractQueue** and implements the **Queue** interface**.**

PriorityQueue is a generic class that has this declaration:

class PriorityQueue<E>

Here, E specifies the type of objects stored in the queue. PriorityQueues are dynamic, growing as necessary.

PriorityQueue defines the six constructors shown here:

PriorityQueue( )

PriorityQueue(int *capacity)*

PriorityQueue(int *capacity, Comparator<? super E> comp)*

PriorityQueue(Collection<? extends E> *c)*

PriorityQueue(PriorityQueue<? extends E> *c)*

PriorityQueue(SortedSet<? extends E> *c)*

The first constructor builds an empty queue. Its starting capacity is 11.

The second constructor builds a queue that has the specified initial capacity.

The third constructor builds a queue with the specified capacity and comparator.

The last three constructors create queues that are initialized with the elements of the collection passed in *c.*

✓ ***In all cases, the capacity grows* automatically as elements are added.**

Accessing a Collection via an Iterator

Method	Description
boolean hasNext( )	Returns **true** if there are more elements. Otherwise, returns **false**.
E next( )	Returns the next element. Throws **NoSuchElementException** if there is not a next element.
void remove( )	Removes the current element. Throws **IllegalStateException** if an attempt is made to call **remove( )** that is not preceded by a call to **next( )**.

Using an Iterator

Before you can access a collection through an iterator, you must obtain one.

Collection classes provides an **iterator( )** method that **returns** an iterator to the start of the collection.

By using this iterator object, you can access each element in the collection one element at a time.

In general, to use an iterator to cycle through the contents of a collection, follow these steps:

1. Obtain an iterator to the start of the collection by calling the collection's **iterator( )** method.

2. Set up a loop that makes a call to **hasNext( ).** Have the loop iterate as long as **hasNext( )** returns **true.**

3. Within the loop, obtain each element by calling **next( ).**

For collections that implement **List,** you can also obtain an iterator by calling **listIterator( ).**

A list iterator gives you the ability to access the collection in either the **forward** or **backward** direction and lets you modify an element.

ListIterator is available only to those collections that implement the **List interface.**

```java
// Demonstrate iterators.
import java.util.*;
class IteratorDemo {
    public static void main(String args[]) {
        // Create an array list.
        ArrayList<String> al = new ArrayList<String>();
        // Add elements to the array list.
        al.add("C");
        al.add("A");
        al.add("E");
        al.add("B");
        al.add("D");
        al.add("F");
```

```java
// Use iterator to display contents of al.
System.out.print("Original contents of al: ");
Iterator<String> itr = al.iterator();
while(itr.hasNext()) {
    String element = itr.next();
    System.out.print(element + " ");
}
System.out.println();
```

The output is shown here:
Original contents of al: C A E B D F

```
//................ Modify objects being iterated...........
ListIterator<String> litr = al.listIterator();
while(litr.hasNext()) {
        String element = litr.next();
        litr.set(element + "+");
}
System.out.print("Modified contents of al: ");
itr = al.iterator();
while(itr.hasNext()) {
        String element = itr.next();
        System.out.print(element + " ");
}
System.out.println();
```

Modified contents of al: C+ A+ E+ B+ D+ F+

```
        // Now, display the list backwards.
        System.out.print("Modified list backwards: ");
        while(litr.hasPrevious()) {
            String element = litr.previous();
            System.out.print(element + " ");
        }
        System.out.println();
    }
}
```

Modified list backwards: F+ D+ B+ E+ A+ C+

Stack class

Stack is a subclass of **Vector** that implements a standard **last-in, first-out stack**.

It is a **generic** class **in java.util** package , that is one can create stack of different types of **objects**.

Stack includes all the methods defined by Vector and adds several of its own, shown in following table:

Note: Vector implements a dynamic array.

It is similar to **ArrayList,** but differences:

Vector is synchronized.

It contains many legacy methods that are not part of the Collections Framework.

Vector reengineered to extend AbstractList and to implement the **List.**

Method	Description
boolean empty( )	Returns **true** if the stack is empty, and returns **false** if the stack contains elements.
E peek( )	Returns the element on the top of the stack, but does not remove it.
E pop( )	Returns the element on the top of the stack, removing it in the process.
E push(E *element*)	Pushes *element* onto the stack. *element* is also returned.
int search(Object *element*)	Searches for *element* in the stack. If found, its offset from the top of the stack is returned. Otherwise, –1 is returned.

EmptyStackException is thrown if **pop( )** called when the invoking stack is empty.

You can use **peek( )** to return, but not remove, the top object.

The **empty( )** method returns true if nothing is on the stack.

The **search( )** method determines whether an object exists on the stack and returns the number of pops (i.e. **offset**) that are required to bring it to the top of stack.

```
// Demonstrate the Stack class.
import java.util.*;
class StackDemo {
    static void showpush(Stack<Integer> st, int a) {
        st.push(a);
        System.out.println("push(" + a + ")");
        System.out.println("stack: " + st);
    }
    static void showpop(Stack<Integer> st) {
        System.out.print("pop -> ");
        Integer a = st.pop();
        System.out.println(a);
        System.out.println("stack: " + st);
    }
```

```
public static void main(String args[]) {
    Stack<Integer> st = new Stack<Integer>();
    System.out.println("stack: " + st);
    showpush(st, 42);
    showpush(st, 66);
    showpush(st, 99);
    showpop(st);
    showpop(st);
    showpop(st);
    try {
            showpop(st);
        }
    catch (EmptyStackException e) {
            System.out.println(“Empty stack");
        }
    } //End of main()
}  //End of class StackDemo
```

stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack

Collection algorithms

- The collections framework also provides polymorphic versions of algorithms you can run on collections.
  - Sorting
  - Shuffling
  - Routine Data Manipulation
    - Reverse
    - File copy
    - etc.
  - Searching
    - Binary Search
  - Composition
    - Frequency
    - Disjoint
  - Finding extreme values
    - Min
    - Max

475 page no.