9 TEXT-BASED APPLICATIONS

Objectives

- Write a program that uses command-line arguments and system properties
- Write a program that reads from *standard input*
- Write a program that can create, read, and write files
- Write a program that uses sets and lists
- Write a program to iterate over a collection

Command-Line Arguments

- Any Java technology applications can use command-line arguments
- These string arguments are placed on the command lien to launch the Java interpreter, after the class name:

```
java TestArgs arg1 arg2 “another arg”
```

- Each command-line arguments is placed in the args array that is passed to the static `main` method:

```
public static void main(String[] args)
```

```
public class TestArgs {
   public static void main(String[] args) {
      for ( int i=0 ; i<args.length ; i++ ) {
         System.out.println("args[" + i +
                            "] is '" +
                            args[i] + "'") ;
      }
   }
}
```

java TestArgs arg1 arg2 "another arg"

System Properties

- System properties is a feature that replaces the concept of *environment variables* (which is platform-specific)
- The `System.getProperties` method returns a `Proeprties` object
  - System.getProperties()
  - System.getProperties(String)
  - System.getProperties(String,String)
- The `getProperty` method returns a `String` representing the value of the named property.
- Use the –D option to include a new property.

The Properties Class

- The `Properties` class implements a mapping of names to values (a `String` to `String` map).
- The `propertyNames` method returns an `Enumeration` of all property names.
- The `getProperty` method returns a `String` representing the value of the named property.
- You can also read and write a properties collection into a file using `load` and `store`.

```
import java.util.Properties ;
import java.util.Enumeration ;
public class TestProperties {
   public static void main(String[] args) {
    Properties props = System.getProperties() ;
    Enumeration prop_names = props.propertyNames();
    while( prop_names.hasMoreElements() ){
      String prop_name=(String)prop_names.nextElement() ;
      String property = props.getProperty(prop_name) ;
      System.out.println("property '" + prop_name +
                              "' is '" + property + "'") ;
      }
  }
}
```

property ‘java.runtime.name’ is ‘Java(TM) 2 Runtime Environment, Standard Edition’
property ‘sun.boot.library.path’ is ‘F:\jdk1.3\jre\bin’
property ‘java.vm.version’ is ‘1.3.0-C’
property ‘java.vm.vendor’ is ‘Sun Microsystems Inc.’
property ‘java.vendor.url’ is ‘http://java.sun.com/’
property ‘path.separator’ is ‘;’
property ‘java.vm.name’ is ‘Java HotSpot(TM) Client VM’
property ‘file.encoding.pkg’ is ‘sun.io’
property ‘java.vm.specification.name’ is ‘Java Virtual Machine Specification’
property ‘user.dir’ is ‘C:\’
property ‘java.runtime.version’ is ‘1.3.0-C’
property ‘java.awt.graphicsenv’ is ‘sun.awt.Win32GraphicsEnvironment’
property ‘os.arch’ is ‘x86’
property ‘java.io.tmpdir’ is ‘C:\DOCUME~1\kurt\LOCALS~1\Temp\’
property ‘line.separator’ is ‘ ’
property ‘java.vm.specification.vendor’ is ‘Sun Microsystems Inc.’
property ‘java.awt.fonts’ is ‘’
property ‘os.name’ is ‘Windows 2000’

```
String osName= (System.getProperties()).getProperty("os.name");

switch ( osName.hashCode() ) {
    case -113889189 :
         System.out.println(“Program has not been tested on this os!") ;
     break;
    default:
           System.out.println("Ok.") ;
}
```

Console I/O

- `System.out` allows you to write to “standard output”.
  - It is an object of type `PrintStream`.
- `System.in` allows you to read from “standard input”.
  - It is an object of type `InputStream`.
- `System.err` allows you to write to “standard error”
  - It is an object of type `PrintStream`.

Writing to Standard Output

- The `println` methods print the argument and a newline '\n'.
- The `print` methods print the argument without a newline
- The `print` and `println` methods are overloaded for most primitive types (`boolean`, `char`, `int`, `long`, `float`, and `double`) and for `char[]`, `Object`, and `String`.
- The `print(Object)` and `println(Object)` methods call the `toString()` method on the argument.

Reading From Standard Input

```
import java.io.* ;
public class KeyboardInput {
    public static void main(String[] args) throws IOException {
      String s                                                    ;
      InputStreamReader ir = new InputStreamReader(System.in) ;
      BufferedReader in = new BufferedReader(ir)                  ;
      System.out.println("Enter an integer : ")  ;
     try{

      while( (s = in.readLine()) != null )  )
            System.out.println("Read: "+ s) ;

         in.close();
     }
     catch( IOException e) {
          e.printStackTrace();
     }
    }
}
```

Files and File I/O

- The `java.io` package
- Creating `File` Objects
- Manipulating `File` Objects
- Reading and writing to file streams

Creating a New File Object

- `File myFile ;`
- `myFile = new File(“myfile.txt”) ;`
- `myFile = new File(“MyDocs”,“myfile.txt”) ;`
- Directories are treated just like files in Java; the File class supports methods for retrieving an array of files in the directory
- ```
  File myDir = new File(“MyDocs”) ;
  myFile = new File(myDir,“myfile.txt”) ;
  ```

The class File defines platform-independent methods for manipulating a file maintained by a native file system. However, it does not allow you to access the contents of the file.
  ```

File Tests and Utilities

- `File` names :
  - String getName()
  - String getPath()
  - String getAbsolutePath()
  - String goParent()
  - boolean renameTo(File newName)
- `File` tests :
  - boolean exists()
  - boolean canWrite()
  - boolean canRead()
  - boolean isFile()
  - boolean isDirectory()

- `File` information
  - long lastModified()
  - long length()
  - boolean delete()
- `File` utilities :
  - boolean mkdir()
  - String[] list()

File Stream I/O

- File input:
  - Use the `FileReader`. class to read characters
  - Use the `BufferedReader` class to use the `readLine` method
- File output:
  - Use the `FileWriter` class to write characters
  - Use the `PrintWriter` class to use the `print` and `println` methods

```
import java.io.* ;
public class ReadFile {
    public static void main(String[] args) {
      File file = new File(args[0])                                    ;
      try {
         BufferedReader in = new BufferedReader(new FileReader(file)) ;
         String s ;
         s = in.readLine() ;
         while( s != null ) {
               System.out.println("Read: " + s ) ;
               s = in.readLine() ;
         }
         in.close() ;
      } catch ( FileNotFoundException e1) {
          System.err.println("File not found: " + file);
      } catch ( IOException e2 ) {
          e2.printStackTrace() ;
      }
   }
}
```

```
import java.io.* ;
public class WriteFile {
    public static void main(String[] args) {
      File file = new File(args[0])                                    ;
      try {
         BufferedReader in = new
                        BufferedReader(new InputStreamReader(System.in)) ;
         PrintWriter out = new PrintWriter(new FileWriter(file)) ;
         String s ;
         System.out.print("Enter file text.   ") ;
         System.out.println("[Type ctrl-d (or ctrl-z) to stop.]") ;
         while( (s=in.readLine()) != null ) {
               out.println(s) ;
         }
         in.close() ;
         out.close();
      } catch ( IOException e) {
        e.printStackTrace() ;
      }
    }
}
```

The Math Class

- The `Math` class contains a group of static math functions
  - Truncation: `ceil`, `floor`, and `round`
  - Variations on max, min, and abs
  - Trigonometry: `sin`, `cos`, `tan`, `asin`, `atan`, `toDegrees`, and `toRadians`
  - Logarithms: `log` and `exp`
  - Others: `sqrt`, `pow`, and `random`
  - Canstants: `PI` and `E`

The String Class

- `String` objects are immutable sequences of Unicode characters.
- Operations that create new strings: `concat`, `replace`, `substring`, `toLowerCase`, `toUpperCase`, and `trim`
- Search operations: `endsWith`, `startWith`, `indexOf`, and `lastIndexOf`.
- Comparisons: `equals`, `equalsIgnoreCase`, and `compareTo`
- Others: `charAt`, and `length`

Methods That Create New Strings

- `String concat(String s)` – returns a new string consisting of `this` string followed by the s string
- `String replace(char old, char new)` – returns a new string that is copy of `this` string with the `new` string replacing all occurences of the `old` string
- `String substring(int start,int end)` – returns a portion of `this` string starting at the `start` index and ending at `end`.
- `String toLowerCase()` – returns a new string consisting of `this` string converted to lowercase
- `String toUpperCase()` – returns a new string consisting of `this` string converted to uppercase

Search Methods

- `boolean endsWith(String s)` – returns `true` if `this` string ends with s
- `boolean startsWith(String s)` – returns `true` if `this` string starts with s
- `int indexOf(String s)` – returns index within `this` string that starts with s
- `int indexOf(int ch)` – returns index within `this` string of the first occurence of the character ch
- `int indexOf(String s, int offset)` – returns index within `this` string that matches with s starting at `offset`
- `int indexOf(int ch, int offset)` – returns index within `this` string of the first occurence of the character ch starting at `offset`

Comparison Methods

- `boolean equals(String s)` – returns `true` if `this` string is equal to string s
- `boolean equalsIgnoreCase(String s)` – returns `true` if `this` string is equal to (ignoring case) the string s
- `int compareTo(String s)` – performs a lexical comparison between `this` string and s; returns a negative `int` if `this` string is less than s, a positive `int` if `this` string is greater than s, or 0 if the two strings are equal

The StringBuffer Class

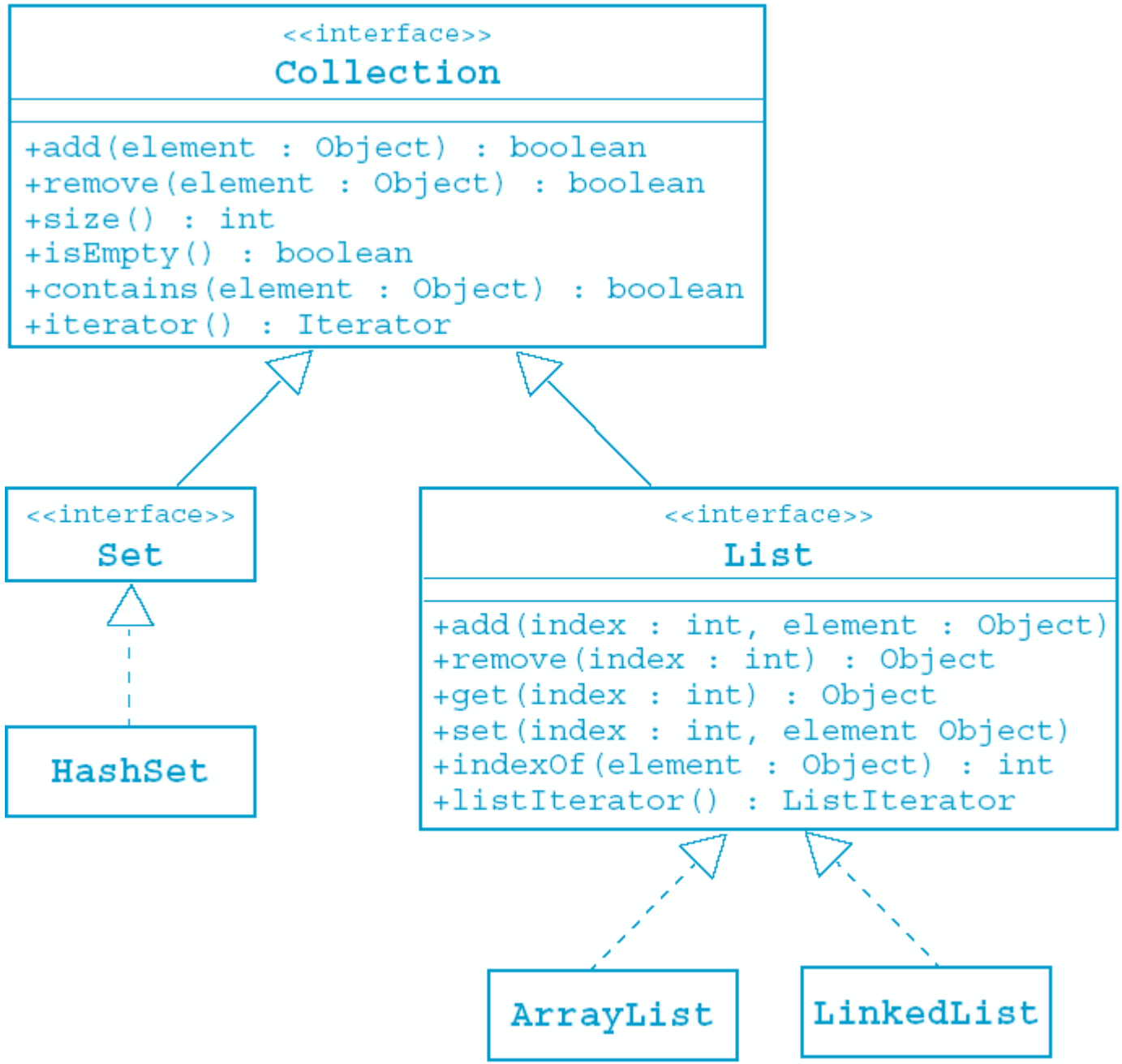
- `StringBuffer` objects are mutable sequences of Unicode characters
- Constructors:
  - `StringBuffer()` – creates an empty buffer
  - `StringBuffer(int capacity)` – creates an empty buffer with a specified initial capacity
  - `StringBuffer(String initialString)` – creates a buffer that initially contains the specified string
- Modification operations: `append`, `insert`, `reverse`, `setCharAt`, and `setLength`

Modifications Methods

- `StringBuffer append(String s)` – Modifies `this` string buffer by appending the s string onto the end of the buffer
- `StringBuffer insert(int offset, String s)` – Modifies `this` string buffer by inserting the s string into the buffer at the specified offset location.
- `StringBuffer reverse()` – Reverses the order of the string buffer
- `void setCharAt(int index, char ch)` – Modifies `this` string buffer by changing the character at the location specified by `index` to the specified character, `ch`.
- `void setLength(int newLength)`

The Collections API

- A *collection* is a single object representing a group of objects known as its elements
- Collection API contains interfaces that group objects as a:
  - `Collection` – A group of objects called elements; any specific ordering and allowance of duplicates is specified by each implementation
  - `Set` – An unordered collection; no duplicates are permitted
  - `List` – An ordered collection; duplicates are permitted


<<interface>>
Collection
+add(element : Object) : boolean
+remove(element : Object) : boolean
+size() : int
+isEmpty() : boolean
+contains(element : Object) : boolean
+iterator() : Iterator
<<interface>>
Set
HashSet
<<interface>>
List
+add(index : int, element : Object)
+remove(index : int) : Object
+get(index : int) : Object
+set(index : int, element Object)
+indexOf(element : Object) : int
+listIterator() : ListIterator
ArrayList
LinkedList

Set Example

```
import java.util.* ;
public class SetExample {
    public static void main(String[] args) {
        Set set = new HashSet();
        set.add("one") ;
        set.add("second") ;
        set.add("3rd") ;
        set.add(new Integer(4)) ;
        set.add(new Float(5.0F)) ;
        set.add("second") ;
        set.add(new Integer(4)) ;
        System.out.println(set) ;
    }
}
```

List Example

```
import java.util.* ;
public class ListExample {
    public static void main(String[] args) {
        List list = new ArrayList();
        list.add(“one”) ;
        list.add(“second”) ;
        list.add(“3rd”) ;
        list.add(new Integer(4)) ;
        list.add(new Float(5.0F)) ;
        list.add(“second”) ;
        list.add(new Integer(4)) ;
        System.out.println(list) ;
    }
}
```

Iterators

- Iteration is the process of retrieving every element in a collection
- An `Iterator` of a `Set` is unordered
- A `ListIterator` of a `List` can be scanned forwards (using the `next` method) or backwards (using the `previous` method):

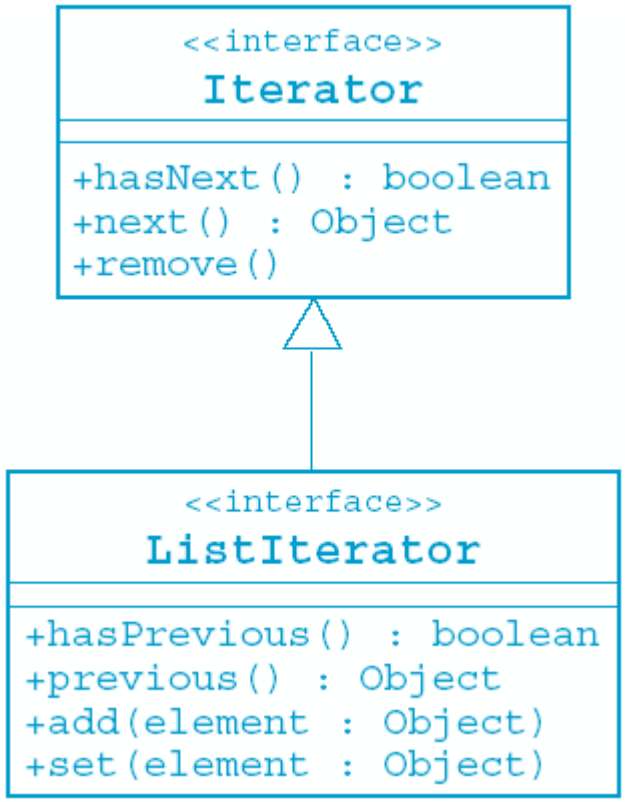
```
List list = new ArrayList() ;

Iterator elements = list.iterator();

while( elements.hasNext() )

   System.out.println( elements.next() )
```

The Iterator Interface Hierarchy



Collections in JDK 1.1

- `Vector` implements the `List` interface
- `Stack` is a subclass of `Vector` and supports the `push`, `pop`, and `peek` methods
- `Hashtable` implements the `Map` interface
- `Enumeration` is a variation on the `Iterator` interface:
  - An enumeration is returned by the `elements` method in `Vector`, `Stack`, and `Hashtable`

10 Building Java GUIs

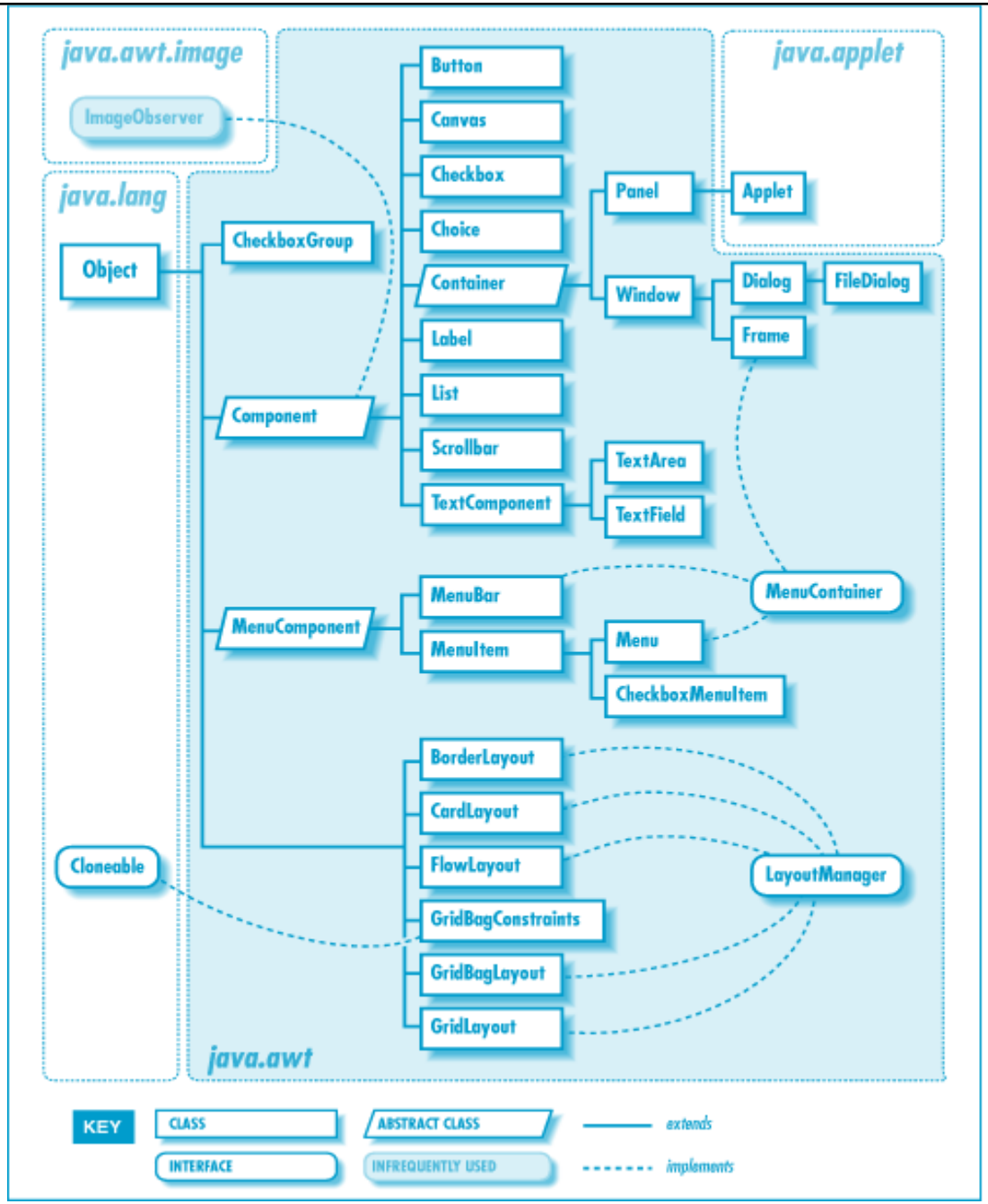
Objectives

- Describe the AWT package and its components
- Define the terms *containers*, *components* and *layout managers*, and how they work together to build a graphical user interface (GUI)
- Use layout managers
- Use the flow, border, grid, and card layout managers to achieve a desired dynamic layout
- Add components to a container
- Use the frame and panel containers appropriately

Abstract Window Toolkit (AWT)

- Provides basic GUI components which are used in all Java applets and applications
- Contains super classes which can be extended and their properties inherited; classes can also be abstract
- Ensures that every GUI component that is displayed on the screen is a subclass of the abstract class `Component` or `MenuComponent`
- Contains `Container` which is an abstract subclass of `Component` and which includes two subclasses:
  - `Panel`
  - `Window`

User-interface classes of AWT Package



Containers

- Add components with the `add()` method
- The two main types of containers are `Window` and `Panel`
- A `Window` is a free floating window on the display. There are two important types of Window:
  - `Frame` – window with a title and corners you can resize
  - `Dialog` – cannot have a menu bar, you cannot resize
- A `Panel` is a container of GUI components that must exist in the context of some other container, such as a window or applet

Positioning Components

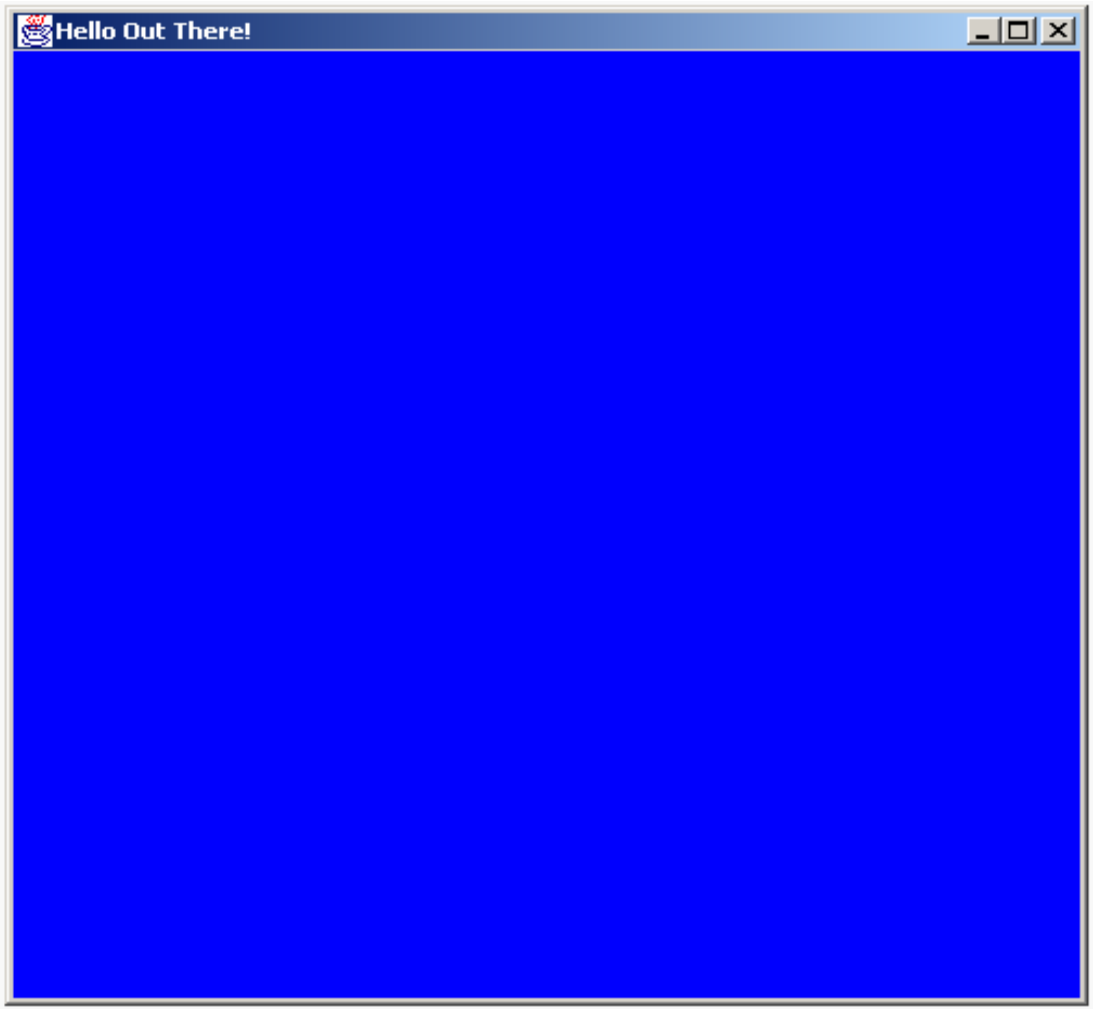
- The position and size of a component in a container is determined by a layout manager.
- You can control the size or position of components by disabling the layout manager.

You must then use `setLocation()`, `setSize()`, or `setBounds()` on components to locate them in the container.

Frames

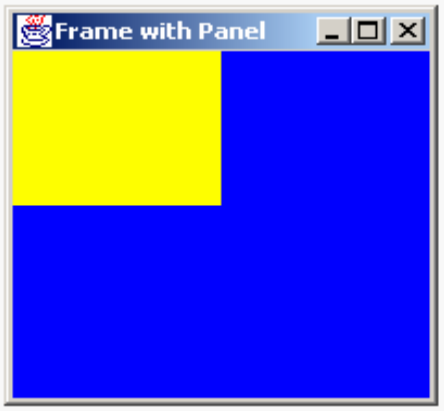
- Are subclasses of `Window`
- Have title and resize corners
- Inherit from Component and add components with the *add* method
- Are initially invisible, use `setVisible(true)` to expose the frame
- Have `BorderLayout` as the default layout manager
- Use the `setLayout` method to change the default layout manager

```
import java.awt.*;
    public class MyFrame extends Frame {
    public static void main (String args[]) {
            MyFrame fr = new MyFrame("Hello Out There!");
            // Component method setSize()
            fr.setSize(500,500);
            fr.setBackground(Color.blue);
            fr.setVisible(true); // Component method show()
    }
    public MyFrame (String str) {
            super(str);
    }
}
```


Hello Out There!

Panels

- Provide a space for components
- Allow subpanels to have their own layout manager
- Add components with the add method




```
public class FrameWithPanel extends Frame {
        public FrameWithPanel (String str) {
                super (str);
        }
        public static void main (String args[]) {
          FrameWithPanel fr =
              new FrameWithPanel ("Frame with Panel");
        Panel pan = new Panel();
        fr.setSize(200,200);
        fr.setBackground(Color.blue);
         fr.setLayout(null); //override default layout manager
        pan.setSize (100,100);
        pan.setBackground(Color.yellow);
        fr.add(pan);
        fr.setVisible(true);
    }
}
```

Container Layout

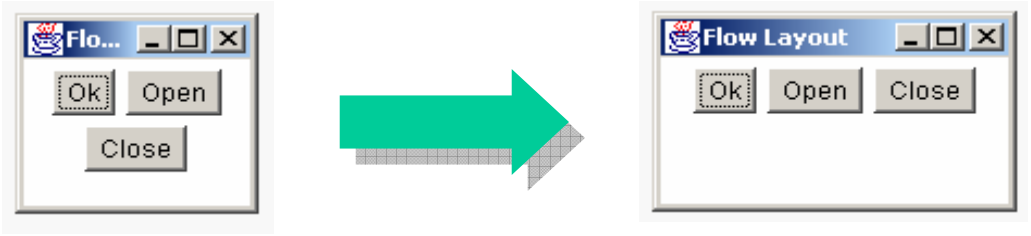
- `FlowLayout`–The default layout manager of Panel and Applet
- `BorderLayout`–The default manager of `Window`, `Dialog`, and `Frame`
- `GridLayout`–A layout manager that provides flexibility for placing components
- `CardLayout`
- `GridBagLayout`

The `FlowLayout` Manager

- Default layout manager for the `Panel` class
- Components are added from left to right
- Default alignment is centered
- Uses components' preferred sizes
- Uses the constructor to tune behavior

```
public class MyFlow {
        private Frame f;
        private Button button1, button2, button3;

        public static void main (String args[]) {
                MyFlow mflow = new MyFlow ();
                mflow.go();
        }
        public void go() {
            f = new Frame ("Flow Layout");
            f.setLayout(new FlowLayout());
            button1 = new Button("Ok");
            button2 = new Button("Open");
            button3 = new Button("Close");
            f.add(button1);
            f.add(button2);
            f.add(button3);
            f.setSize (100,100);
            f.setVisible(true);
        }
}
```

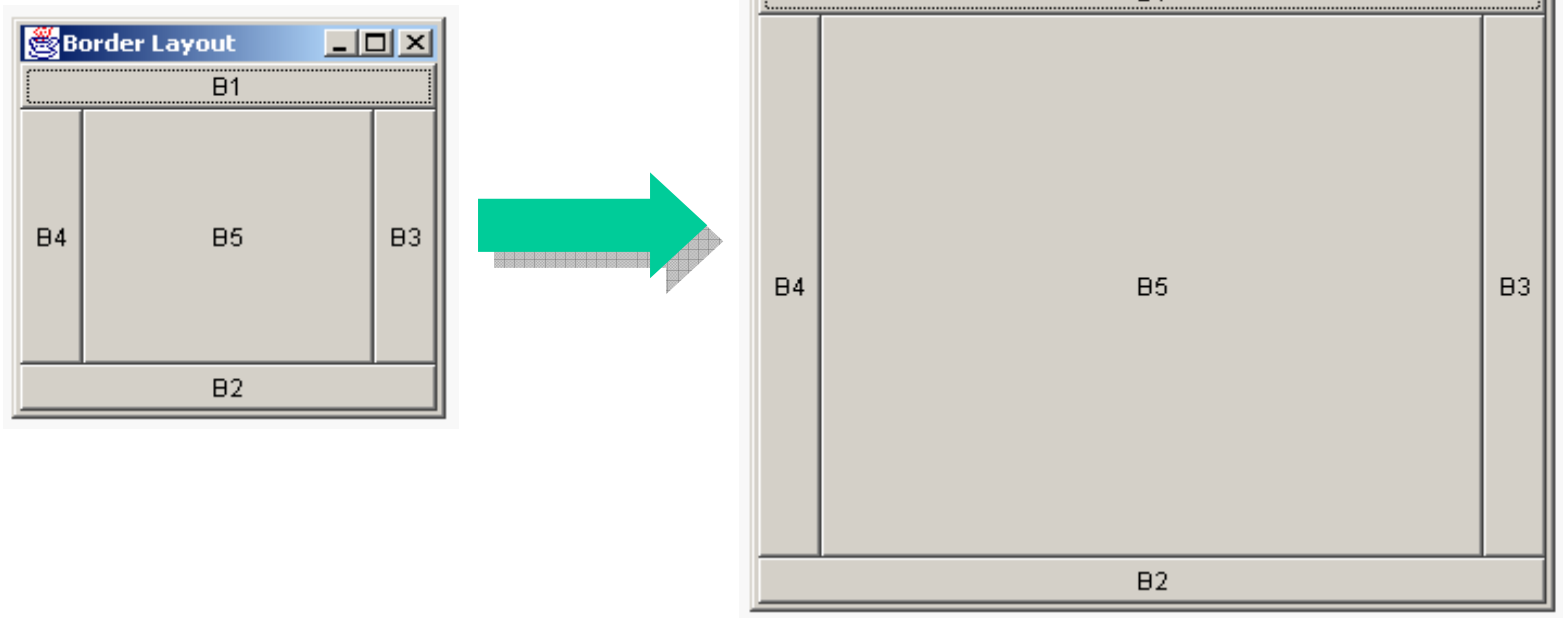


The BorderLayout Manager

- Default layout manager for the `Frame` class
- Components are added to specific regions
- The resizing behavior:
  - North, South, and Center regions adjust horizontally
  - East, West, and Center regions adjust vertically

```
public class BorderExample {
        private Frame f;
        private Button bn, bs, bw, be, bc;
        public static void main(String args[]) {
            BorderExample guiWindow2 = new BorderExample();
            guiWindow2.go();
        }
        public void go() {
            f = new Frame("Border Layout");
            bn  =  new Button("B1");
            bs  =  new Button("B2");
            be  =  new Button("B3");
            bw =  new Button("B4");
            bc  =  new Button("B5");

            f.add(bn, BorderLayout.NORTH);
            f.add(bs, BorderLayout.SOUTH);
            f.add(be, BorderLayout.EAST);
            f.add(bw, BorderLayout.WEST);
            f.add(bc, BorderLayout.CENTER);
            f.setSize (200, 200);
            f.setVisible(true);
          }
}
```

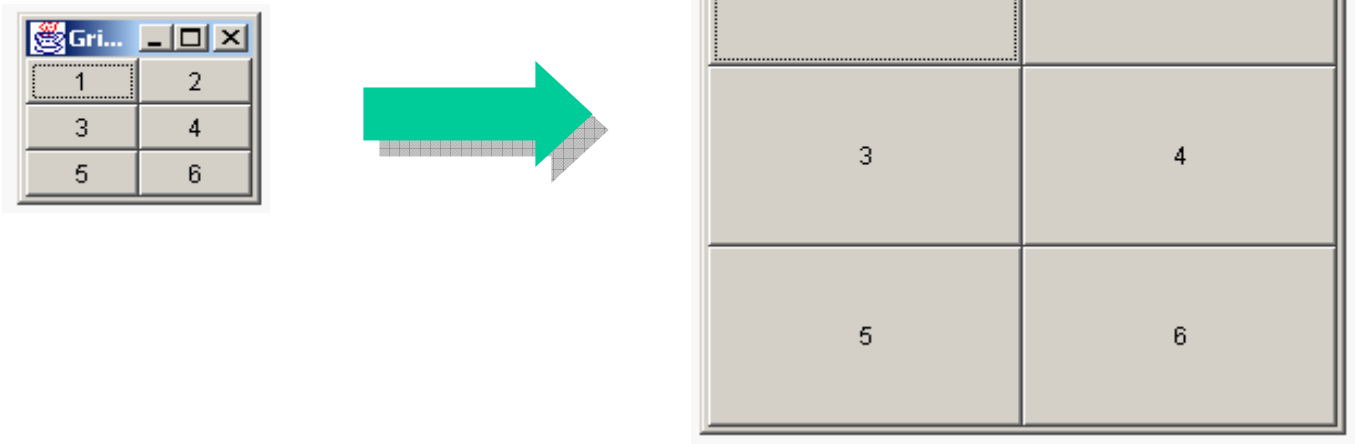

Border Layout
B1
B4
B5
B3
B2
Border Layout
B1
B4
B5
B3
B2

The GridLayout Manager

- Components are added left to right, top to bottom.
- All regions are equally sized.
- The constructor specifies the rows and columns.

```
public class GridExample {
        private Frame f;
        private Button b1, b2, b3, b4, b5, b6;
        public static void main(String args[]) {
                GridExample grid = new GridExample();
                grid.go();
        }
        public void go() {
                f = new Frame("Grid example");
                f.setLayout (new GridLayout (3, 2));
```

```
            b1 = new Button("1");
            b2 = new Button("2");
            b3 = new Button("3");
            b4 = new Button("4");
            b5 = new Button("5");
            b6 = new Button("6");
            f.add(b1);
            f.add(b2);
            f.add(b3);
            f.add(b4);
            f.add(b5);
            f.add(b6);
            f.pack();
            f.setVisible(true);
        }
    }
```


Gri...
1
2
3
4
5
6
Grid example
1
2
3
4
5
6

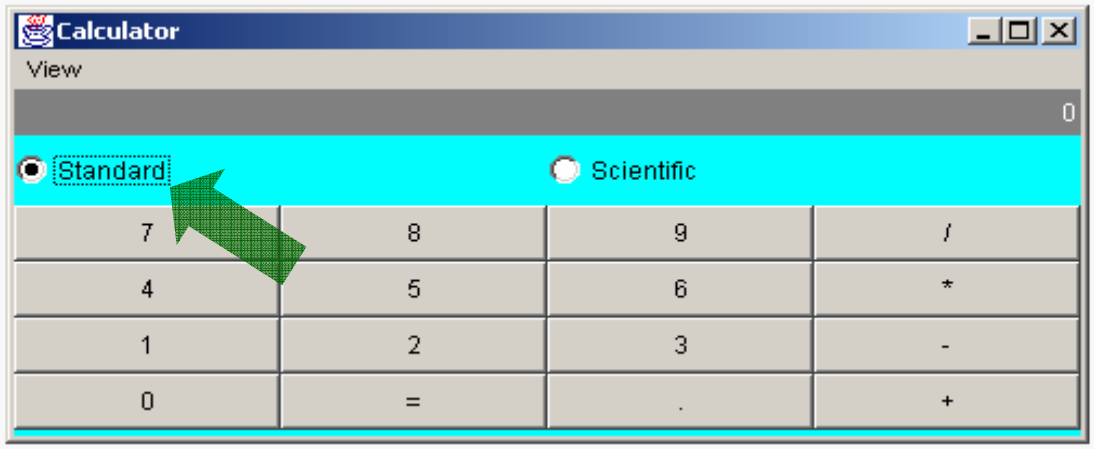
The CardLayout Manager

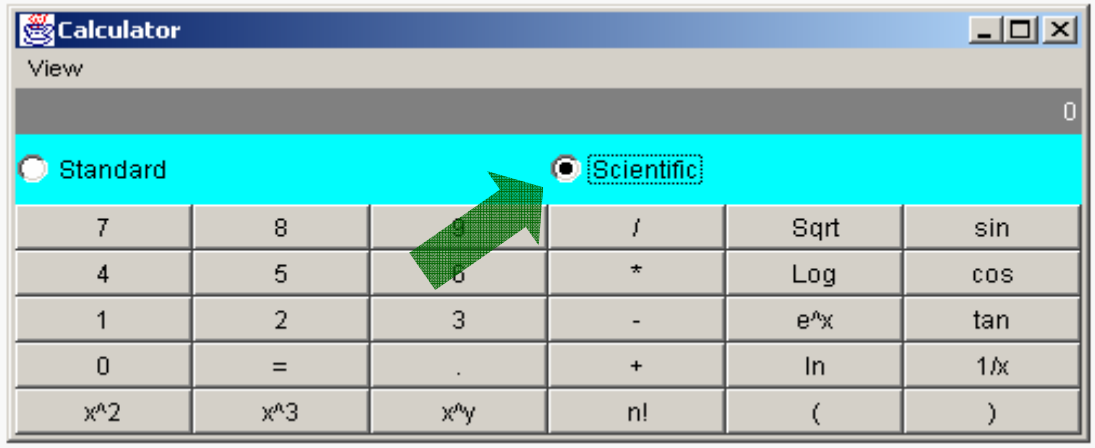
- The `CardLayout` manager arranges components into a "deck" of cards where only the top card is visible.

- Each card is usually a container such as a panel and each card can use any layout manager

```
 // add deck
deck = new Panel() ;
cardManager = new CardLayout() ;
deck.setLayout(cardManager) ;
      // add Card 1
     standardCalculator = new Panel() ;
     standardCalculator.setLayout(new GridLayout(4,4)) ;
     standardButtons = new Button[16] ;
      for(int i=0;i<16;i++)  {
          standardButtons[i] = new Button(standardString[i]) ;
          standardButtons[i].addActionListener(this) ;
          standardCalculator.add(standardButtons[i])   ;
     }
     deck.add(standardCalculator,"standard") ;
```

```
// add Card 2
      scientificCalculator = new Panel() ;
      scientificCalculator.setLayout(new GridLayout(5,6)) ;
      scientificButtons = new Button[30] ;
      for(int i=0;i<30;i++) {
          scientificButtons[i] = new Button(scientificString[i]) ;
          scientificButtons[i].addActionListener(this) ;
          scientificCalculator.add(scientificButtons[i])   ;
       }
      deck.add(scientificCalculator,"scientific")        ;
```


Calculator
View
0
Standard
Scientific
7
8
9
/
4
5
6
*
1
2
3
-
0
=
.
+


Calculator
View
0
Standard
Scientific
7
8
9
/
Sqrt
sin
4
5
6
*
Log
cos
1
2
3
-
e^x
tan
0
=
.
+
ln
1/x
x^2
x^3
x^y
n!
(
)

The GridBagLayout Manager

- A Layout manager similar to `GridLayout`.
- Unlike `GridLayout` each component size can vary and components can be added in any order