1. GUI
   1. **Java Foundation Class (JFC)**: group of features to develop GUIs and adding graphics functionality and interactivity.
   2. Swing GUI Components: javax.swing and javax.swing.event
      1. *Containment hierarchy*: every UI component must be part of a tree of components that has a top-level container as its root; can be contained only once.
      2. **JComponents**: superclass of all J-name components except 3 top-level containers.
         1. Subclasses
            1. **JPanel**
            2. **JScrollPane**: provides scrollable view of a component for large/dynamic sizes.

*Viewport view*: JScrollPane's client, responsible for positioning and sizing the client based on positions of scrollbars, and displaying it.

Change the client dynamically by calling setViewportView().

When the user manipulates the scroll bars in a JScrollPane, the area of the client that is visible changes accordingly.

Scrollbar policy

setHorizontalScrollBarPolicy()

setVerticalScrollBarPolicy()

ScrollPaneConstants

Default policy: scroll bar appears when the viewport is smaller than the client and disappears when the viewport is larger than the client.

VERTICAL\_SCROLLBAR\_AS\_NEEDED

HORIZONTAL\_SCROLLBAR\_AS\_NEEDED

Always display scrollbar: knob disappears if the viewport is large enough to show the whole client.

VERTICAL\_SCROLLBAR\_ALWAYS

HORIZONTAL\_SCROLLBAR\_ALWAYS

Never display scrollbar

VERTICAL\_SCROLLBAR\_NEVER  
HORIZONTAL\_SCROLLBAR\_NEVER

* + - * 1. **JButton**: a clickable button that can display both text and an image.

Text in a different place relative to its image

Underlined letter in each button's text shows the *mnemonic*.

Implement event handling

Normally an action listener which is notified every time the user clicks the button.

* + - * 1. **JTable**
      1. Methods
         1. setToolTipText()
         2. setBorder()
         3. paintComponent()
         4. putClientProperty() and getClientProperty()
         5. Layout

setMinimumSize()

setMaximumSize()

setAlignmentX()

setAlignmentY()

* + - * 1. Key bindings

setMnemonic()

* + - 1. 3 top-level containers: each has a content pane that contains the visible components in that top-level container's UI.
         1. **JFrame**

Root in Swing-based UI

* + - * 1. **JDialog**: independent subwindow used for temporary purposes independent of the main window content.

2 dialogs in UI and applets; root of dialog hierarchy

Subclass of AWT  java.awt.Dialog class.

Adds root pane container and support for default close operation on Dialog.

Usages

Display error message

Display warning

Present images

Directory trees

Swing component classes that directly instantiate and display dialogs

JOptionPane

ProgressMonitor

JColorChooser

JFileChooser

Printing API

Every dialog is dependent on a JFrame component. If the frame is…

destroyed, so are the dependent Dialogs

iconified, its dependent Dialogs also disappear from the screen.

deiconified, its dependent Dialogs return to the screen.

**Modal dialogs**: blocks user input to all other windows in the program when visible.

JOptionPane: container that automatically creates a modal JDialog and add itself to the JDialog's content pane.

Uses

lays out standard dialogs

provides icons

specify the dialog title and text

customizing the button text

customize the components the dialog displays

specify where the dialog should appear onscreen

Icon options

Custom

No icon

4 standard icons (Question, Info, Warning, Error)

Otherwise, use JDialog

Instantiation

Frame argument if used is the frame that the dialog depends upon.

Arguments

Boolean argument

true: modal

false/absent: non-modal

String: specify title

setDefaultCloseOperation()

DO\_NOTHING\_ON\_CLOSE

HIDE\_ON\_CLOSE

DISPOSE\_ON\_CLOSE

* + - * 1. **JApplet**

Root in Swing-based applets

* + - * 1. Optionally add a menu bar to a top-level container
      1. Text
         1. **JTextField**
         2. **JFormattedTextField**
         3. **JPasswordField**
         4. **JTextArea**: providing a component to display multiple lines of text, and optionally allows user to edit the text.

Unless the user has moved the **caret** (insertion point) by clicking or dragging in the text area, the text area automatically scrolls to the appended text with JScrollPane.

Can force text area to scroll to the bottom.

textArea.setCaretPosition(textArea.getDocument().getLength());

Customizing

Arguments used by JScrollPane to determine text area.

Rows

Columns

Text is editable by default

setEditable(false) if needed

Still selectable and able to copy

Font and color

setFont(new Font("Serif", Font.ITALIC, 16));

Determine how text area wraps lines and the number of characters per tab

textArea.setLineWrap(true);

textArea.setWrapStyleWord(true);

When NOT to use a JTextArea

Single line of input from a user

Use JTextField.

Text area displaying multiple fonts or other styles

Use JEditorPane or JTextPane.

Displayed text with a limited length and is never edited by the user.

Use JLabel.

* + - * 1. JEditorPane
        2. JTextPane
    1. Pluggable Look-and-Feel Support
       1. Set ComponentUI using UIManager.setLookAndFeel()
    2. Accessibility API
    3. Java 2D API
    4. Internationalization
    5. Drag and drop: set a component's transfer handler.
    6. Double buffering: smooths onscreen painting.
  1. **Root pane**: each top-level container relies on a reclusive intermediate that manages the content pane and the menu bar in addition to a couple of other containers.
     1. **Glass pane**:hidden, by default, if made visible it's like a sheet of glass over all the other parts of the root pane.
        1. Completely transparent unless you implement the glass pane's paintComponent().
        2. Can intercept input events for the root pane.
     2. **Layered pane**: serves to position its contents (content pane and optional menu bar).
        1. Can hold other components in a specified Z order for information.
        2. Contents
           1. **Content pane**: container of the root pane's visible components, excluding the menu.
           2. **Optional menu bar**:home for the root pane's container's menus.
        3. If the container has a menu bar, you generally use the container's setJMenuBar method to put the menu bar in the appropriate place
  2. Adding Components
     1. Referencing content pane
        1. Using JFrame (default layout manager – BorderLayout)

JFrame frame = new JFrame();

Label label = new Label();

frame.getContentPane().add(label); //Must be typecast; otherwise create

//own component

* + - 1. Using JPanel (default layout manager – FlowLayout)

JPanel contentPane = new JPanel(new BorderLayout());

contentPane.setBorder(*someBorder*);

contentPane.add(*someComponent*, BorderLayout.CENTER);

contentPane.add(*anotherComponent*, BorderLayout.PAGE\_END);

*topLevelContainer*.setContentPane(contentPane);

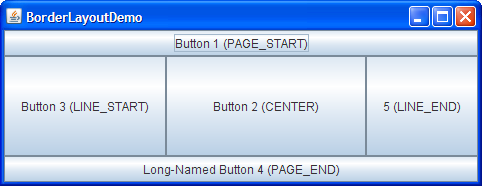
* + 1. Adding Menu Bar (usually only in applets and UI)
       1. Steps
          1. Create a JMenuBar object
          2. Populate it with menus
          3. Call setJMenuBar()
       2. Example

JFrame frame = new JFrame();

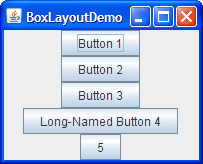
JMenuBar greenMenuBar = new JMenuBar();

frame.setJMenuBar(greenMenuBar);

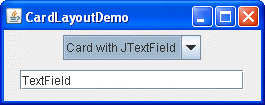
* 1. LayoutManagers
     1. **BorderLayout**: arranges the components into 5 areas.



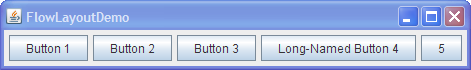
* + - 1. PAGE\_START or NORTH
      2. PAGE\_END or SOUTH
      3. LINE\_START or EAST
      4. LINE\_END or WEST
      5. CENTER
    1. **BoxLayout**: puts components in a single row or column, respecting maximum sizes, and allows for aligning components.



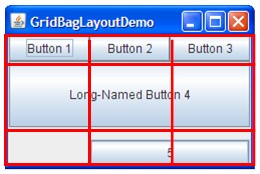
* + 1. **CardLayout**: lets you implement an area that contains different components at different times, often controlled by a **combo box**, with its state determining which panel to display.



* + - 1. Alternatively use a tabbed pane.
    1. **FlowLayout**: lays out components in a single row in a specified order or from left to right, starting a new row if the container is insufficiently wide.



* + 1. **GridBagLayout**: places components in a grid; allows them to span multiple rows or columns of cells, defined by preferred sizes.



* + - 1. One of most flexible but complex layout managers.
      2. GridBagConstraints: specifies size and position characteristic constraints to each component.
         1. Use Container.add(), passing it a GridBagConstraints object
         2. Example 1

JPanel pane = new JPanel(new GridBagLayout());

OR

pane.setLayout(new GridBagLayout());

OR

GridBagLayout gridBagLayout = new GridBagLayout();

pane.setLayout(gridBagLayout);

GridBagConstraints c = new GridBagConstraints();

pane.add(theComponent, c);

* + - * 1. Example 2

// X is the column

// Y is the row

// W is the width in cells

// H is the height in cells

// aContainer is the container the component is added to

// aComponent is the component being added to the container

private void addComponent( int x, int y, int w, int h, Container aContainer, Component aComponent )

{

constraints.gridx = x;

constraints.gridy = y;

constraints.gridwidth = w;

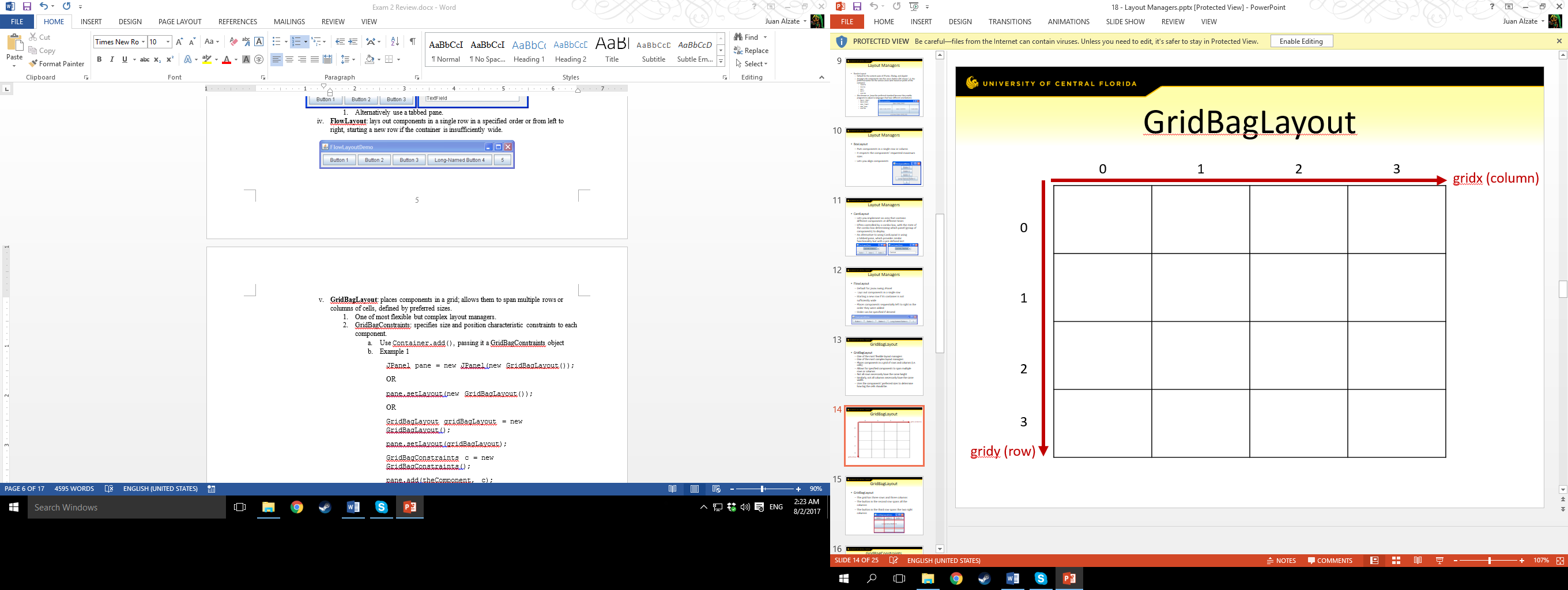
constraints.gridheight = h;

gridBagLayout.setConstraints( aComponent, constraints );

aContainer.add( aComponent );

}

* + - * 1. Instance variables



gridx: specifies the cell containing the leading edge of the component's display area, where the first cell in a row has gridx == 0.

gridy: specifies the cell at the top of the component's display area, where the topmost cell has gridy == 0.

gridwidth: specifies the number of cells in a row for the component's display area.

gridheight: specifies the number of cells in a column for the component's display area.

weightx: specifies how to distribute extra horizontal space.

weighty: specifies how to distribute extra vertical space.

anchor: used when the component is smaller than its display area.

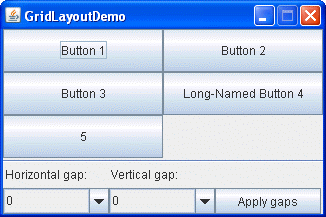
fill: used when the component's display area is larger than the component's requested size.

insets: specifies the external padding of the component, the minimum amount of space between the component and the edges of its display area.

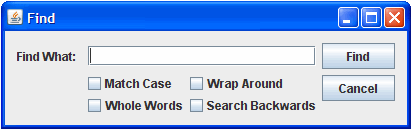
ipadx: specifies the internal padding of the component, how much space to add to the minimum width of the component.

ipady: specifies the internal padding, that is, how much space to add to the minimum height of the component.

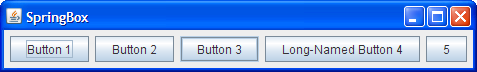
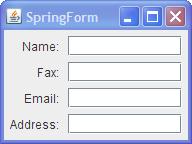
* + 1. **GridLayout**: arranges components into rows and columns equal in size.



* + 1. **GroupLayout**: developed for use by GUI builder tools, but can be used manually.



* + - 1. Works with horizontal and vertical layouts separately.
      2. Each component needs to be defined twice in layout.
    1. **SpringLayout**: developed for use by GUI builder tools; specify precise relationships between the edges of components under its control; lays out the children of its associated container according to a set of constraints.

* + - 1. Low-level but flexible
  1. **ActionListeners**: event handlers to define what should be done when a user performs certain operation.
     1. Events: an actionPerformed message is sent to all action listeners that are registered on the relevant component.
        1. Click button
        2. Choose menu item
        3. Press enter in text field
     2. Example

public class MyEventHandler implements ActionListener {

//...

public void actionPerformed(ActionEvent e) {

//code that reacts to the action

}

//...

}

someComponent.addActionListener(instanceOfMyClass);

* + 1. ActionEvent Class
       1. String getActionCommand() and getActionCommand(): return or set string associated with action event.
       2. Object getSource(): returns Object that fired event.
       3. int getModifiers(): return integer representing the modifier keys the user was pressing when the action event occurred.
          1. Constants

SHIFT\_MASK

CTRL\_MASK

META\_MASK

ALT\_MASK

* + 1. **Timer**
       1. Types
          1. javax.swing.Timer: fires one or more action events after a time delay.

Used with UI development

Perform task once, after delay

Perform task repeatedly

* + - * 1. java.util.Timer
      1. Specify an ActionListener to be notified by actionPerformed() when the timer goes off and the **speed** (number of milliseconds between time firings).
      2. Modifications
         1. Pause before ActionListener is invoked: setInitialDelay(pause)
         2. Execute action once: setRepeats(false)
         3. Start: start()
         4. Stop: stop()
      3. Example: periodically update a component.
         1. Timer timer = new Timer(speed, actionListener);
         2. timer.setInitialDelay(pause);
         3. timer.start();
         4. timer.stop();
      4. Speed is represented in ms.

1. **Exceptions**: “exceptional events”; problems that occur during program execution that disrupt the normal flow of instructions, causing the program to terminate abnormally.
   1. Reasons
      1. A user has entered invalid data.
      2. A file that needs to be opened cannot be found.
      3. A network connection has been lost in the middle of communications or the JVM has run out of memory.
   2. Types
      1. **Checked (compile-time) exceptions**: occur at compile time; cannot simply be ignored.
         1. Since the FileReader methods **read()** and **close()** throw IOException, compiler notifies to handle IOException, along with FileNotFoundException.
      2. **Unchecked (runtime) exceptions**: occur at the time of execution, and ignored at compilation time.
         1. Programming bugs
            1. Logic errors
            2. Improper API use
      3. **Errors**: not exceptions, but severe and unrecoverable failures that arise beyond the control of the user or the programmer (or Java programs) and which are ignored at compile time.
         1. Stack overflow
         2. Run out of memory
   3. Hierarchy
      1. java.lang.Throwable
         1. Exception
            1. IOException
            2. RuntimeException
         2. Error
   4. Exception Methods
      1. **public String getMessage()**: returns a detailed message about the exception that has occurred, initialized in the Throwable constructor.
      2. **public Throwable getCause()**: returns the cause of the exception as represented by a Throwable object.
      3. **public String toString()**: returns the name of the class concatenated with the result of getMessage().
      4. **public void printStackTrace()**: prints the result of toString() along with the stack trace to System.err, the error output stream.
      5. **public StackTraceElement [] getStackTrace()**: returns an array containing each element on the stack trace. The element at index 0 represents the top of the call stack, and the last element in the array represents the method at the bottom of the call stack.
      6. **public Throwable fillInStackTrace()**: fills the stack trace of this Throwable object with the current stack trace, adding to any previous information in the stack trace.
   5. Exception Handling
      1. **Exception object**: contains information about error (type and program state).
         1. Creating custom exceptions
            1. Must be of child Throwable
            2. To be automatically enforced by Handle or Declare Rule, extend Exception.
            3. Runtime exception: extend RunTimeException.
      2. **Throwing an exception**: the hand-off of the created object to the runtime system.
         1. Runtime system attempts to find within the **call stack**, an ordered list of methods that were called prior to the thrown exception to handle it.
         2. If a method does not handle a checked exception, it must declare it using the throws keyword.
            1. Used to postpone the handling of a checked exception
         3. You can throw exceptions by using the throw keyword with the exceptions separated by commas.
            1. Used to invoke an exception explicitly.
      3. Try, catch, finally
         1. **Try block**: encloses the code that might throw and exception.
            1. Requires catch or try statement
         2. **Catch block(s)**: follow immediately follow the closing curly brace of a try block; handles the type of exception indicated by its argument.
            1. Requires try statement
         3. **Finally block**: ALWAYS executed when the try block exits unless the JVM exits, a thread is interrupted or killed; useful for cleaning up code or preventing resource leaks.
            1. Not required.
         4. No code must be present between try, catch, or finally statements.
         5. Example

try {//…//}

catch(FileNotFoundException e){//…///}

catch(IOException e){//…///}

catch(IOException|SQLException ex){

logger.log(ex);

throw ex;

}

finally {

if (out != null) System.out.println("Closing PrintWriter"); out.close();

else System.out.println("PrintWriter not open");

}

1. **Logger**: objects where applications make logging calls.
   1. Characteristics
      1. Organized in hierarchical namespace.
         1. By default all, Loggers also send their output to their parent Logger.
      2. Each Logger keeps track of a set of output Handlers.
      3. Loggers may also be configured to ignore Handlers higher up the tree.
   2. Allocate LogRecord objects which are passed to Handler objects for publication.
      1. When necessary to publish a LogRecord externally, a Handler can use a Formatter to localize and format the message before publishing it to an I/O stream.
   3. May use
      1. **Logging levels**: rough guide of 7 levels to the importance and urgency of a log message, present in both Loggers and Handlers.
         1. SEVERE: serious failure.
         2. WARNING: potential failure.
         3. INFO: informational messages.
         4. CONFIG: static configuration messages.
         5. FINE: tracing info.
         6. FINER: fairly detailed tracing info.
         7. FINEST: highly detailed tracing message.
      2. OFF: can turn off logging.
      3. ALL: can enable logging of all messages.
      4. **Filters**: to decide if interested in a particular LogRecord.
         1. Order of filtering: Logger then Handler.
2. Generics
   1. **Collection (Container)**: groups multiple elements into a single unit; used to store, manipulate, and communicate aggregate data.
      1. Natural groups
         1. Poker hand
         2. Mail folder
         3. Telephone directory
   2. **Collections framework**: unified architecture for representing and manipulating collections.
      1. All contain
         1. **Interfaces**: allow collections to be manipulated independently of the details of their representation; on OOP interfaces generally form a hierarchy.
            1. Abstract data types that represent collections
         2. **Implementations**: concrete implementations of the collection interfaces; reusable data structures.
         3. **Algorithms:** methods that perform useful computations, such as searching and sorting, on objects that implement collection interfaces.
            1. *Polymorphic*: the same method can be used on many different implementations of the appropriate collection interface; reusable functionality.
      2. Benefits
         1. Reduce programming effort
         2. Increase program quality and speed
         3. Allows interoperability among unrelated APIs
         4. Reduces effort to learn and to use new APIs
         5. Reduces effort to design new APIs
         6. Fosters software reuse
      3. Hierarchy
         1. **Collection**: represents a group of objects known as its *elements*; used to pass collections around and to manipulate them when maximum generality is desired.
            1. **Set**: collection that cannot contain duplicate elements.

Models mathematical set abstraction

the cards comprising a poker hand

the courses making up a student's schedule

the processes running on a machine

**SortedSet**: Set that maintains elements in ascending orders.

Several additional operations to take advantage of ordering.

Used for naturally-ordered sets

Word lists

Membership rolls

* + - * 1. **List (sequence)**: ordered collection with duplicate elements.

Control over where in the list each element is inserted

Can access elements by their integer index

* + - * 1. **Queue**: collection used to hold multiple elements prior to processing.

Provides additional operations

Insertion

Extraction

Inspection

Order

Typically: orders elements in a **FIFO** (first-in, first-out) manner.

All new elements inserted at the tail.

**Priority queues**: order elements according to a supplied comparator or the elements' natural ordering.

Head of the queue is the element that would be removed by a call to remove() or poll().

* + - * 1. **Deque**: collection used to hold multiple elements prior to processing.

Provides additional operations

Insertion

Extraction

Inspection

Order

FIFO

**LIFO** (last-in, first-out)

All new elements can be inserted, retrieved and removed at both ends.

* + - * 1. **Map**: maps keys to values without duplicates.

**SortedMap**: maintains mappings in ascending key order.

Used for naturally-ordered collections of key/value pairs

Dictionaries

Telephone directories

* + 1. Collections Interfaces: contain methods that perform…
       1. Basic operations
          1. int size()
          2. boolean isEmpty()
          3. boolean contains(Object element)
          4. boolean add(E element)
          5. boolean remove(Object element)
          6. Iterator<E> iterator()
       2. Bulk operations: perform an operation on an entire collection.
          1. containsAll(): returns true if the target Collection contains all of the elements in the specified Collection
          2. addAll(): adds all of the elements in the specified Collection to the target Collection
          3. removeAll(): removes from the target Collection all of its elements that are also contained in the specified Collection
          4. retainAll(): removes from the target Collection all its elements that are *not* also contained in the specified Collection. That is, it retains only those elements in the targetCollection that are also contained in the specified Collection.
          5. clear(): removes all elements from the Collection
          6. addAll(), removeAll(), and retainAll(): methods all return true if the target Collection was modified in the process of executing the operation
       3. As a simple example of the power of bulk operations, consider the following idiom to remove *all* instances of a specified element, e, from a Collection, c.
          1. c.removeAll(Collections.singleton(e));
          2. More specifically, suppose you want to remove all of the null elements from a Collection.
          3. c.removeAll(Collections.singleton(null));

This idiom uses Collections.singleton, which is a static factory method that returns an immutable Set containing only the specified element

* + 1. Traversing Collections: 3 ways
       1. **Aggregate operations**: often used in conjunction with lambda expressions to make programming more expressive, using less lines of code.
          1. **Lambda expressions**: expressions enable a developer to treat functionality as method argument, or code as data.
          2. Example

myShapesCollection.stream()

.filter(e -> e.getColor() == Color.RED)

.forEach(e -> System.out.println(e.getName()));

* + - * 1. Can request parallel stream if collection is large enough and computer has enough cores.

myShapesCollection.parallelStream()

.filter(e -> e.getColor() == Color.RED)

.forEach(e -> System.out.println(e.getName()));

String joined = elements.stream()

.map(Object::toString)

.collect(Collectors.joining(", "));

int total = employees.stream(

.collect(Collectors.summingInt(Employee::getSalary)));

* + - 1. **For-each constructs**: construct allows a developer to concisely traverse a collection or array using a for loop.
         1. Example

for (Object o : collection)

System.out.println(o)

* + - 1. **Iterators**: enables a developer to traverse through a collection and to remove elements from the collection selectively, if desired.
         1. Get an interator for a collection by calling its iterator method

public interface Iterator<E> {

boolean hasNext();

E next();

void remove(); //optional

}

* + - * 1. Methods

hasNext(): returns true if the iteration has more elements.

next(): returns the next element in the iteration.

remove(): removes the last element that was returned by next from the underlying Collection; only safe way to modify collection during iteration.

* + - * 1. Use Iterator instead of the for-each construct when you need to:

Remove the current element

for-each construct hides the iterator, so cannot call method remove().

for-each construct is not usable for filtering

Iterate over multiple collections in parallel

* + - * 1. How to use an Iterator to filter an arbitrary Collection — that is, traverse the collection removing specific elements

static void filter(Collection<?> c) {

for (Iterator<?> it = c.iterator(); it.hasNext(); )

if (!cond(it.next()))

it.remove();

}

* + 1. **Array Operations**
       1. toArray methods are provided as a bridge between collections and older APIs that expect arrays on input.
       2. Allow the contents of a Collection to be translated into an array
       3. Simple form with no arguments: creates a new array of Object.
       4. More complex form: allows the caller to provide an array or to choose the runtime type of the output array.
       5. Example
          1. Suppose that c is a Collection, the following code dumps the contents of c into a newly allocated array of Object whose length is identical to the number of elements in c

Object[] a = c.toArray();

* + - * 1. If  Collection c is known to contain only the following code dumps the contents of c into a newly allocated array of String whose length is identical to the number of elements in c

String[] a = c.toArray(new String[0]);

* 1. **Type parameters**: interfaces and classes used as parameters when defining classes, interfaces, and methods to re-use same code with different inputs
     1. Formal parameters: use values.
     2. Type parameters: use types.
     3. Benefits
        1. Stronger type checks at compile time
        2. Elimination of casts
        3. Implement generic algorithms that work on collections of different types
           1. Customizable
           2. Type safe
           3. Easier to read
     4. Format: class name <T1, T2, … Tn> {/\*…\*/}
        1. Type variables: T1, T2, … Tn
           1. **Non-primitive**

Class

Interface

Array

Another type

* + - 1. <>: delimit parameter section
      2. All occurrences of object are replaced by T
    1. Naming conventions
       1. **E**: element (used extensively by the Java Collections Framework)
       2. **K**: key
       3. **N**: number
       4. **T**: type
       5. **V**: value
    2. Invoking and Instantiating
       1. *Generic type invocation*: replaces T with a concrete value.
          1. Box<Integer> integerBox;
          2. Pass non-instantiated *parameterized* *type argument* to class itself.
       2. Instantiating
          1. Box<Integer> integerBox = new Box<Integer>();
          2. Java SE 7: can replace type arguments required to invoke constructor of generic class with empty <> as long as compiler can infer type arguments.
  1. **Raw type**: name of generic class or interface without any type arguments.
     1. Example

public class Box<T> {

public void set(T t) { /\* ... \*/ }

// ...

}

* + 1. Create parameterized type of Box<T> by supplying actual type parameter for formal type parameter T.
       1. Box<Integer> intBox = new Box<>();
    2. Otherwise, create raw type.
       1. Box rawBox = new Box();

1. **Polymorphism**: when an organism has many forms/stages; when a subclass can define its own unique behaviors, but share some of the functionality of the parent class.
   1. Java does not permit extension of more than one class is to avoid the issues of ***multiple inheritance of state*** (inheriting fields from multiple classes).
      1. Problem: methods or constructors from different superclasses instantiate the same field
         1. Which method or constructor will take precedence?
      2. Solution: extend 1 class and implement multiple interfaces.
   2. ***Multiple inheritance of implementation***: ability to inherit methods from multiple classes.
      1. Problems
         1. Name conflicts
         2. Ambiguity
         3. Compilers of programming languages that support this type of multiple inheritance encounter superclasses that contain methods with the same name, may not be able to determine which member or method to access or invoke
         4. Programmers can unintentionally introduce a name conflict by adding a new method to a superclass
   3. ***Multiple inheritance of type***: a class can implement multiple interfaces, and an object can have multiple types (a type of its own class, and the types of all interfaces that the class implements).
      1. Allowed by Java
      2. Class can inherit different implementations of a method defined (as default or static) in the interfaces that it extends.
   4. Overriding/Hiding methods: hiding a static method vs. overriding an instance method
      1. Overridden instance method that gets invoked: the one in the subclass
      2. Hidden static method that gets invoked: depends on whether it is invoked from the superclass or the subclass.
      3. Interface methods: default methods and abstract methods in interfaces are inherited like instance methods
         1. When the supertypes of a class or interface provide multiple default methods with the same signature, inheritance rules are followed to resolve the name conflict.
            1. Instance methods are preferred over interface default methods
            2. Methods that are already overridden by other candidates are ignored, when supertypes share a common ancestor.
2. Abstract
   1. **Abstract superclasses**: abstract classes cannot be used to create objects; only used as superclasses in inheritance.
      1. Incomplete
         1. Subclasses must declare missing implementations for all of the abstract methods in order to create a **concrete class**.
         2. Otherwise, the subclass must also be declared abstract.
      2. Class members
         1. May include **abstract methods**: declared without implementation or braces and followed by a semicolon.
         2. May have static fields and methods used with a class reference
         3. Can declare…
            1. Non-static and final fields
            2. public, protected, and private **concrete methods**
   2. Abstract Class vs. Interface
      1. Similarities
         1. Cannot instantiate them
         2. May contain a mix of methods declared with or without an implementation
      2. Only in interfaces
         1. All fields are automatically public, static, and final
         2. All methods that you declare or define (as default methods) are public
         3. Can implement any number of interfaces
            1. Can only extend 1 abstract class
         4. Must implement ALL of the interface’s methods.
            1. Not true of abstract classes.
      3. When to use abstract classes
         1. Want to share code among several closely-related classes
         2. Expect that classes that extend the abstract class have many common methods or fields, or require access modifiers other than public (such as protected and private).
         3. Want to declare non-static or non-final fields.
            1. Define methods that can access and modify the state of the object to which they belong
      4. When to use interfaces
         1. Expect that unrelated classes would implement the interface
         2. Want to specify the behavior of a particular data type, but not concerned about who implements its behavior
         3. Want to take advantage of multiple inheritance of type.
   3. JDK Abstract Classes
      1. **AbstractMap**: part of Collections framework.
         1. Subclasses
            1. **HashMap**
            2. **TreeMap**
            3. **ConcurrentHashMap**
         2. Shared methods
            1. get()
            2. put()
            3. isEmpty()
            4. containsKey()
            5. containsValue()
      2. **GraphObjects**: circles, rectangles, lines, Bezier curves, and other graphic objects.
         1. Behaviors and states: usually the same, but others require different implementations.
            1. position
            2. orientation
            3. line color
            4. fill color
            5. moveTo()
            6. rotate()
            7. resize()
            8. draw()
         2. Must be able to draw or resize themselves.
            1. Differ in how so; use abstract superclass.
   4. **Final classes**: can declare a class or some or all of a class’s methods as final such that they cannot be overwritten.
      1. Final methods: used when it is critical that the implementation remains unchanged.
         1. Methods called from constructors
      2. Final classes: CANNOT be subclassed.
         1. String class