

CS062

DATA STRUCTURES AND ADVANCED PROGRAMMING

4: The Catch-All Java Lecture



Tom Yeh
he/him/his

Class News

- ▶ Start assignment early
- ▶ Learning communities will be posted tomorrow morning (Friday)

Wrap up Lecture 3: Interfaces, and Generics

- ▶ Interfaces
- ▶ Generics

Interfaces

- ▶ Methods form an object's **interface** with the outside world
- ▶ Interface = a group of related methods with empty bodies
- ▶ Contracts of what a class must do, not how to do it, abstracting from implementation.
- ▶ Central concept in OOP.
- ▶ In Java, an interface is a reference type (like a class), that contains only constants, method signatures, default methods, and static methods.
- ▶ A class that implements an interface is obliged to implement its methods.
- ▶ Method bodies exist only for default methods and static methods.
- ▶ Interfaces cannot be instantiated (no **new** keyword). They can only be **implemented** by classes or **extended** by other interfaces.

Example

```
public interface Moveable{
    int turn(Direction direction, double radius, double speed);

    default int stop(){        // Method bodies only exist for default and static methods
        speed=0;
    }
}

public class Car implements Moveable{
    int turn(Direction direction, double radius, double speed){
        //code goes here
    }
}

public class Bicycle implements Moveable{
    int turn(Direction direction, double radius, double speed){
        //code goes here
    }
}
```

Interfaces

- ▶ A class can implement multiple interfaces.
 - ▶ `class A implements Interface1, Interface2{...}`
- ▶ An interface can extend multiple interfaces.
 - ▶ `public interface GroupedInterface extends Interface1, Interface2{...}`

Lecture 3: Inheritance, Interfaces, and Generics

- ▶ Inheritance
- ▶ Interfaces
- ▶ Generics

Generics

- ▶ Compile-time errors can be easier to fix than run-time errors.
- ▶ Java introduced **generics** (similar to templates in C++) to help move more bugs to compile-time (easier to debug!), eliminate casting, and improve abstraction. E.g.,

```
List list = new ArrayList();  
list.add("hello");  
String s = (String) list.get(0);
```

Is now:

```
List<String> list = new ArrayList<String>();  
list.add("hello");  
String s = list.get(0);    // no cast
```

- ▶ Generics enable types (**classes and interfaces**) to be used as parameters when defining classes, interfaces, and methods.
- ▶ Type parameters allow you to re-use the same code with different inputs
- ▶ Similar to parameters where the inputs are values, the inputs to type parameters are types

Formal and actual type parameters

```
public interface List <E> {  
    void add(E x);  
    Iterator<E> iterator();  
}
```

Formal type parameters



```
public interface Iterator<E> {  
    E next();  
    boolean hasNext();  
}
```

- ▶ In the invocation (e.g., `List<Integer>`) all occurrences of the formal type parameters are replaced by the **actual type argument** (e.g., `Integer`).
- ▶ Similar to how an argument replaces the parameters in a method when it's called, the actual type arguments replace the generic `E` (formal type)

Generic classes

`class` name <T1, T2, ..., Tn> {...}

- ▶ A type variable can be any non-primitive type (class, interface, array)
- ▶ E: element (common in data structures), T: type, K: key, V: value, N: number, etc.

```
/**  
 * Generic version of the Box class.  
 * https://docs.oracle.com/javase/tutorial/java/generics/types.html  
 * @param <T> the type of the value being boxed  
 */
```

```
public class Box<T> {  
    private T t;  
  
    public void set(T t) { this.t = t; }  
    public T get() { return t; }  
}
```

- ▶ Invocation: `Box<Integer> integerBox = new Box<Integer>();`

Multiple Type Parameters Example

```
public interface Pair<K, V> {  
    public K getKey();  
    public V getValue();  
}
```

```
public class OrderedPair<K, V> implements Pair<K, V> {  
    private K key;  
    private V value;  
  
    public OrderedPair(K key, V value) {  
        this.key = key;  
        this.value = value;  
    }  
}
```

```
Pair<String, Integer> p1 = new OrderedPair<String, Integer>("Even", 8);
```

```
OrderedPair<String, Box<Integer>> p = new OrderedPair<String, Box<Integer>>("primes", new  
Box<Integer>(...));
```

Generic methods

`modifier (static) <T1, T2, ..., Tn> return-type name(list of type parameters){...}}`

- ▶ The type parameter's scope is limited to the method which is declared.
- ▶ Static, non-static generic methods, generic class constructors are allowed.
- ▶ **Type inference**: allows you to invoke a generic method as an ordinary method, without specifying a type between angle brackets.
- ▶ E.g., `className/objectName.genericMethod(arguments);`

Example

- ▶ Generic method that swaps the elements of an array at two specified indices.

```
public static <T> void swap(T[] a, int i, int j) {  
    T temp = a[i];  
    a[i] = a[j];  
    a[j] = temp;  
}
```

Readings:

- ▶ Oracle's guides:
 - ▶ Interfaces and Inheritance: <https://docs.oracle.com/javase/tutorial/java/landl/index.html>
 - ▶ Generics: <https://docs.oracle.com/javase/tutorial/java/generics/index.html>
<https://docs.oracle.com/javase/tutorial/extra/generics/intro.html>
- ▶ Textbook:
 - ▶ Pages 100-104, 122
- ▶ Textbook Website:
 - ▶ Generics: <https://algs4.cs.princeton.edu/13stacks/>

Practice Problems:

- ▶ If you want more practice with hiding vs overriding:
<http://javabypatel.blogspot.com/2016/04/java-interview-questions.html>

Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

What is a package?

- ▶ A grouping of related classes and interfaces that provides access protection and name space management.
- ▶ e.g., `java.lang` for fundamental classes or `java.io` for classes related to reading input and writing output.
- ▶ Packages correspond to folders/directories.
- ▶ A package creates a new namespace, prevent conflicts with type names in other packages
- ▶ Lower-case names.
- ▶ `package` whatevername; at top of file.
- ▶ `import` `graphics.*`; for including all classes/interfaces.
- ▶ or `import` `graphics.Circle`; for more specific access.

Access modifiers

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
No modifier	Y	Y	N	N
private	Y	N	N	N

Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Exceptions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events



Java Documentation Generation System

- Reads JavaDoc comments and gives HTML pages
- JavaDoc comment = description written in HTML + tags
- Enclosed in `/**` `*/`
- Must precede class, variable, constructor or method declaration
- For class:
 - **@author** author name - classes and interfaces
 - **@version** date - classes and interfaces
- For method:
 - **@param** param name and description - methods and constructors
 - **@return** value returned, if any - methods
 - **@throws** description of any exceptions thrown - methods

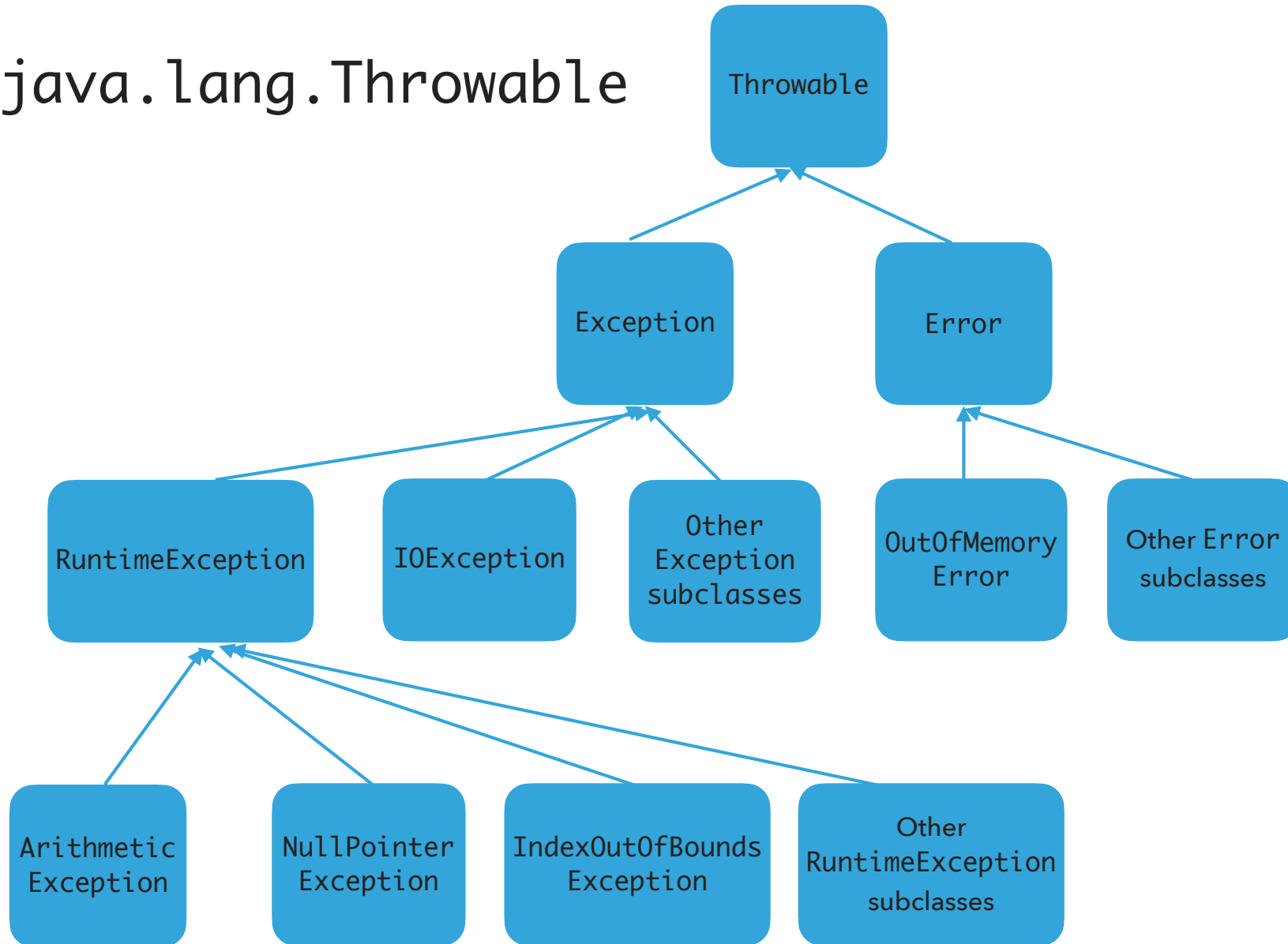
Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

Exceptions are exceptional or unwanted events

- ▶ That is operations that disrupt the normal flow of the program.
 - ▶ E.g., divide a number by zero, run out of memory, ask for a file that does not exist, etc.
- ▶ When an error occurs within a method, the method **throws** an **exception object** that contains its name, type, and state of program.
- ▶ The runtime system looks for something to handle the exception among the **call stack**, the list of methods called (in reverse order) by `main` to reach the error.
- ▶ The exception handler **catches** the exception. If no appropriate handler, the program terminates.

java.lang.Throwable



Three major types of exception classes

- ▶ Error: rare internal system errors that an application cannot recover from.
 - ▶ Typically not caught and program has to terminate.
 - ▶ e.g., `java.lang.OutOfMemoryError` or `java.lang.StackOverflowError`
- ▶ Exception: errors caused by program and external circumstances.
 - ▶ Can be caught and handled.
 - ▶ e.g., `java.io.Exception`
- ▶ RuntimeException: programming errors that can occur in any Java method.
 - ▶ Method not required to declare that it throws any of the exception.
 - ▶ e.g., `java.lang.IndexOutOfBoundsException`, `java.lang.NullPointerException`, `java.lang.ArithmeticException`
- ▶ **Unchecked exceptions**: Error and RuntimeException and subclasses.
- ▶ **Checked exceptions**: All other exceptions - programmer has to check and deal with them.

Handling exceptions

- ▶ Three operations:
 - ▶ Declaring an exception
 - ▶ Throwing an exception
 - ▶ Catching an exception

```
method1(){
```

```
    try {  
        method2();  
    } catch (Exception e) {  
        System.err.println(e.getMessage());  
    }  
}
```

CATCH EXCEPTION

```
method2() throws Exception{
```

DECLARE EXCEPTION

```
    if(some error) {  
        throw new Exception();  
    }  
}
```

THROW EXCEPTION

Declaring exceptions

- ▶ Every method must state the types of *checked* exceptions it might throw in the method header so that the caller of the method is informed of the exception.
 - ▶ System errors and runtime exceptions can happen to any code, therefore Java does not require explicit declaration of unchecked exceptions.
- ▶ `public void exceptionalMethod() throws IOException{`
- ▶ `throws`: the method might throw an exception. Can also throw multiple exceptions, separated by comma.

Throwing exceptions

- ▶ If an error is detected, then the program can throw an exception.
 - ▶ e.g., you have asked for age and the user gave you a string. You can throw an `IllegalArgumentException`.
- ▶ `throw new IllegalArgumentException("Wrong argument");`
 - ▶ The argument in the constructor is called the exception message. You can access it by invoking `getMessage()`.
- ▶ `throws` **FOR DECLARING AN EXCEPTION**, `throw` **TO THROW AN EXCEPTION**

Catching exceptions

- ▶ An exception can be caught and handled in a try-catch block.

```
method(){  
    try {  
        statements; //statements that could thrown exception  
    } catch (Exception1 e1) {  
        //handle e1;  
    }  
    catch (Exception2 e2) {  
        //handle e2;  
    }  
}
```

- ▶ If no exception is thrown, then the catch blocks are skipped.
- ▶ If an exception is thrown, the execution of the try block ends at the responsible statement.
- ▶ The order of catch blocks is important. A compile error will result if a catch block for a superclass type appears before a catch block for a subclass. E.g., `catch(Exception ex)` followed by `catch(RuntimeException ex)` won't compile.
- ▶ If a method declares a checked exception (e.g., `void p1() throws IOException`) and you invoke it, you have to enclose it in a try catch block or declare to throw the exception in the calling method (e.g., `try{ p1();} catch (IOException e){...}`).

finally block

- ▶ Used when you want to execute some code regardless of whether an exception occurs or is caught

```
method(){  
    try {  
        statements; //statements that could throw exception  
    } catch (Exception1 e) {  
        //handle e; catch is optional.  
    }  
    finally{  
        //statements that are executed no matter what;  
    }  
}
```

- ▶ The finally block will execute no matter what. Even after a return.

```
/**
 * Illustrates try,catch, finally blocks
 * @author https://docs.oracle.com/javase/tutorial/essential/exceptions/putItTogether.html
 *
 */
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {
    // Note: This class will not compile yet.

    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers() {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }

    public void writelist() {
        PrintWriter out = null;

        try {
            System.out.println("Entering" + " try statement");

            out = new PrintWriter(new FileWriter("OutFile.txt"));
            for (int i = 0; i < SIZE; i++) {
                out.println("Value at: " + i + " = " + list.get(i));
            }
        } catch (IndexOutOfBoundsException e) {
            System.err.println("Caught IndexOutOfBoundsException: " + e.getMessage());
        } catch (IOException e) {
            System.err.println("Caught IOException: " + e.getMessage());
        } finally {
            if (out != null) {
                System.out.println("Closing PrintWriter");
                out.close();
            } else {
                System.out.println("PrintWriter not open");
            }
        }
    }
}
```

Practice Time

- ▶ 1. Is there anything wrong with this exception handler?

```
try {  
  
} catch (Exception e) {  
  
} catch (ArithmeticException a) {  
  
}
```

Answers

- ▶ 1. The ordering matters! The second handler can never be reached and the code won't compile.

Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

TEXT

Stopped here!

Pre and post conditions

- ▶ **Pre-condition**: Specification of what must be true for method to work properly.
- ▶ **Post-condition**: Specification of what must be true at end of method if precondition held before execution.

Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

I/O streams

- ▶ **Input stream**: a sequence of data into the program.
- ▶ **Output stream**: a sequence of data out of the program.
- ▶ Stream sources and destinations include disk files, keyboard, peripherals, memory arrays, other programs, etc.
- ▶ Data stored in variables, objects and data structures are temporary and lost when the program terminates. Streams allow us to save them in files, e.g., on disk or CD (!)
- ▶ Streams can support different kinds of data: bytes, principles, characters, objects, etc.

Files

- ▶ Every file is placed in a directory in the file system.
- ▶ **Absolute file name**: the file name with its complete path and drive letter.
 - ▶ e.g., on Windows: `C:\temp\somefile.txt`
 - ▶ On Mac/Unix: `/home/temp/somefile.txt`
- ▶ `File`: contains methods for obtaining file properties, renaming, and deleting files. Not for reading/writing!
- ▶ **CAUTION: DIRECTORY SEPARATOR IN WINDOWS IS \, WHICH IS SPECIAL CHARACTER IN JAVA. SHOULD BE "\\” INSTEAD.**

TEXT I/O

```
/**
 * Demonstrates File class and its operations.
 * @author https://liveexample.pearsoncmg.com/html/TestFileClass.html
 *
 */

import java.io.File;
import java.util.Date;

public class TestFile {
    public static void main(String[] args) {
        File file = new File("some.text");
        System.out.println("Does it exist? " + file.exists());
        System.out.println("The file has " + file.length() + " bytes");
        System.out.println("Can it be read? " + file.canRead());
        System.out.println("Can it be written? " + file.canWrite());
        System.out.println("Is it a directory? " + file.isDirectory());
        System.out.println("Is it a file? " + file.isFile());
        System.out.println("Is it absolute? " + file.isAbsolute());
        System.out.println("Is it hidden? " + file.isHidden());
        System.out.println("Absolute path is " + file.getAbsolutePath());
        System.out.println("Last modified on " + new Date(file.lastModified()));
    }
}
```

Writing data to a text file

- ▶ `PrintWriter output = new PrintWriter(new File("filename"));`
- ▶ New file will be created. If already exists, discard.
- ▶ Invoking the constructor may throw an I/O Exception...
- ▶ `output.print` and `output.println` work with Strings, and primitives.
- ▶ Always close a stream!

TEXT I/O

```
/**
 * Demonstrates how to write to text file.
 * @author https://liveexample.pearsoncmg.com/html/WriteData.html
 *
 */

import java.io.File;
import java.io.IOException;
import java.io.PrintWriter;

public class WriteData {
    public static void main(String[] args) {

        PrintWriter output = null;
        try {
            output = new PrintWriter(new File("addresses.txt"));
            // Write formatted output to the file
            output.print("Alexandra Papoutsaki ");
            output.println(222);
            output.print("Tom Yeh ");
            output.println(128);

        } catch (IOException e) {
            System.err.println(e.getMessage());
        } finally {
            if (output != null)
                output.close();
        }
    }
}
```


Reading data from a text file

- ▶ `java.util.Scanner` reads Strings and primitives.
- ▶ Breaks input into tokens, demoted by whitespaces.
- ▶ To read from keyboard: `Scanner input = new Scanner(System.in);`
- ▶ To read from file: `Scanner input = new Scanner(new File("filename"));`
- ▶ Need to close stream as before.
- ▶ `hasNext()` tells us if there are more tokens in the stream. `next()` returns one token at a time.
 - ▶ Variations of `next` are `nextLine()`, `nextByte()`, `nextShort()`, etc.

TEXT I/O

```
/**
 * Demonstrates how to read data from a text file.
 * @author https://liveexample.pearsoncmg.com/html/ReadData.html
 *
 */

import java.io.File;
import java.io.IOException;
import java.util.Scanner;

public class ReadData {
    public static void main(String[] args) {

        Scanner input = null;
        // Create a Scanner for the file
        try {
            input = new Scanner(new File("addresses.txt"));

            // Read data from a file
            while (input.hasNext()) {
                String firstName = input.next();
                String lastName = input.next();
                int room = input.nextInt();
                System.out.println(firstName + " " + lastName + " " + room);
            }
        } catch (IOException e) {
            System.err.println(e.getMessage());
        } finally {
            if (input != null)
                input.close();
        }
    }
}
```

Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

GUIs

- ▶ **AWT**: The Abstract Windowing Toolkit is found in the package `java.awt`
 - ▶ Heavyweight components.
 - ▶ Implemented with native code written for that particular computer.
 - ▶ The AWT library was written in six weeks!
- ▶ **Swing**: Java 1.2 extended AWT with the `javax.swing` package.
 - ▶ Lightweight components.
 - ▶ Written in Java.

JFrame

- ▶ `javax.swing.JFrame` inherits from `java.awt.Frame`
- ▶ The outermost container in an application.
- ▶ To display a window in Java:
 - ▶ Create a class that extends `JFrame`.
 - ▶ Set the size.
 - ▶ Set the location.
 - ▶ Set it visible.

JFrame

```
import javax.swing.JFrame;

public class MyFirstGUI extends JFrame {

    public MyFirstGUI() {
        super("First Frame");
        setSize(500, 300);
        setLocation(100, 100);
        setVisible(true);
    }

    public static void main(String[] args) {
        MyFirstGUI mfgui = new MyFirstGUI();
    }
}
```



Closing a GUI

- ▶ The default operation of the quit button is to set the visibility to false. The program does not terminate!
- ▶ `setDefaultCloseOperation` can be used to control this behavior.
- ▶ `mfgui.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`
- ▶ More options (hide, do nothing, etc).

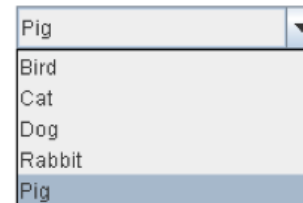
Basic components



[JButton](#)



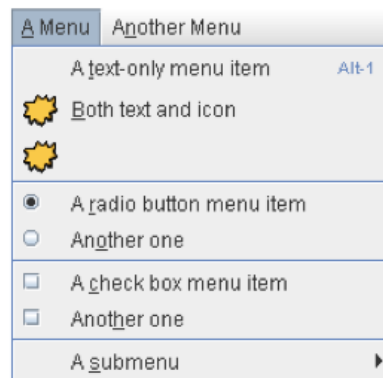
[JCheckBox](#)



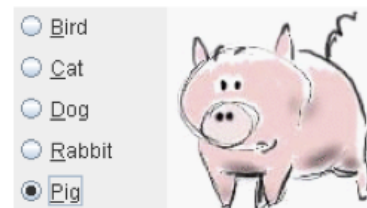
[JComboBox](#)



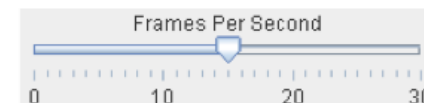
[JList](#)



[JMenu](#)



[JRadioButton](#)



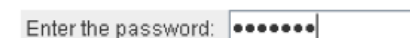
[JSlider](#)



[JSpinner](#)

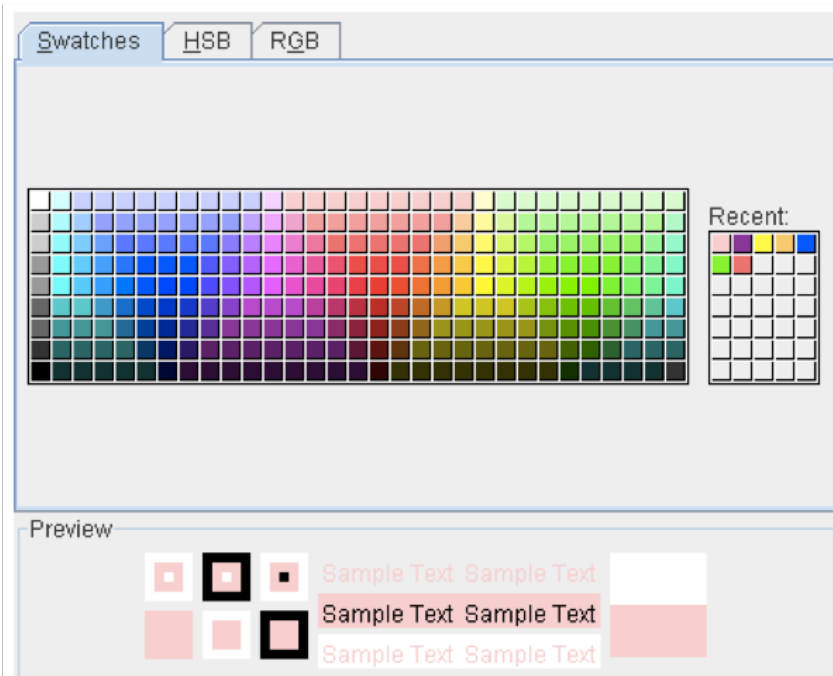


[JTextField](#)

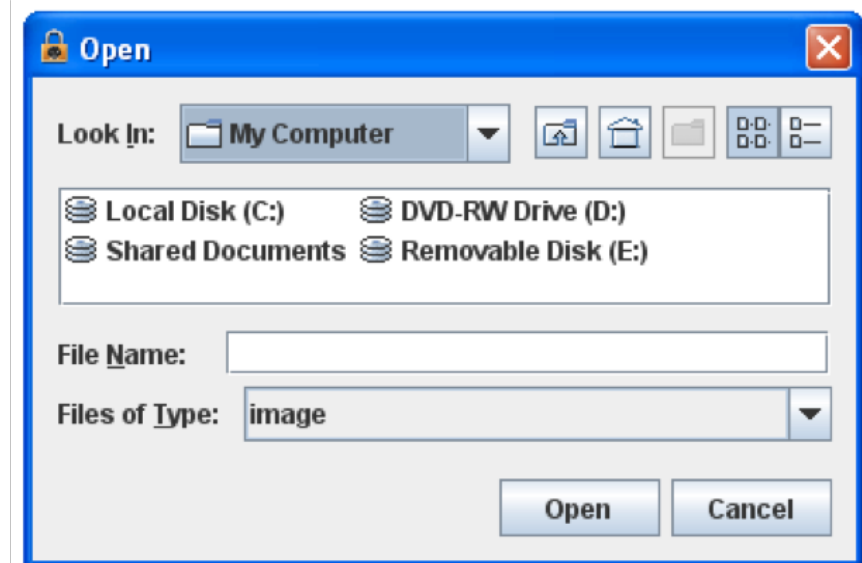


[JPasswordField](#)

Interactive displays



[JColorChooser](#)



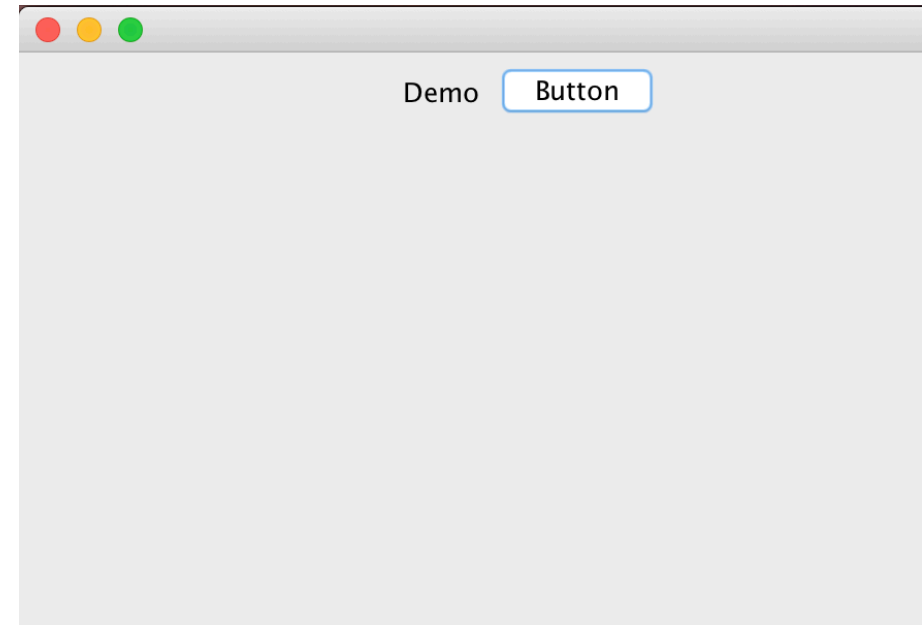
[JFileChooser](#)

Adding JComponents to JFrame

```
import java.awt.Container;
import java.awt.FlowLayout;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;

public class GUIDemo extends JFrame {
    public GUIDemo() {
        // Container cp = getContentPane();
        // cp.setLayout(new FlowLayout());
        // cp.add(new JLabel("Demo"));
        // cp.add(new JButton("Button"));
        JPanel mainPanel = new JPanel(new FlowLayout());
        mainPanel.add(new JLabel("Demo"));
        mainPanel.add(new JButton("Button"));
        setContentPane(mainPanel);
        setSize(500, 300);
        setVisible(true);
    }

    public static void main(String[] args) {
        GUIDemo gd = new GUIDemo();
        gd.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
}
```



Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

Java Graphics

- ▶ Create arbitrary objects you want to draw:
 - ▶ `Rectangle2D.Double`, `Line.Double`, etc.
 - ▶ Constructors take x, y coordinates and dimensions, but don't actually draw items.
- ▶ All drawing takes place in `paint` method using a "graphics content".
- ▶ Triggered implicitly by uncovering window or explicitly by calling the `repaint` method.
 - ▶ Adds repaint event to draw queue and eventually draws it.

Graphics context

- ▶ All drawing is done in `paint` method of component.
- ▶ `public void paint (Graphics g)`
- ▶ `g` is a graphics context provided by the system.
- ▶ “pen” that does the drawing.
- ▶ You call `repaint()` not `paint()`.
- ▶ Need to import classes from `java.awt.*`, `java.geom.*`, `javax.swing.*`
- ▶ See `MyGraphicsDemo`.

General graphics applications

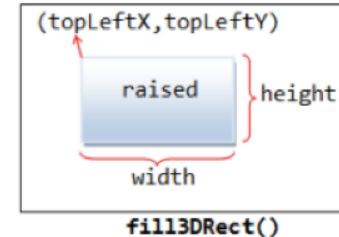
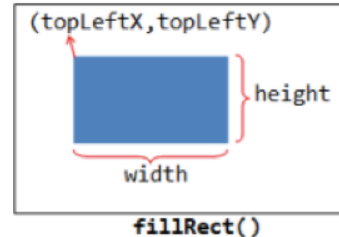
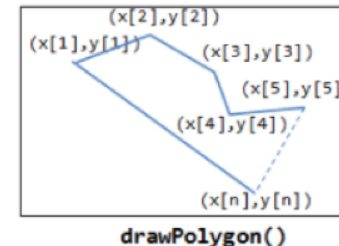
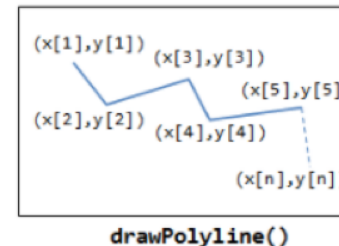
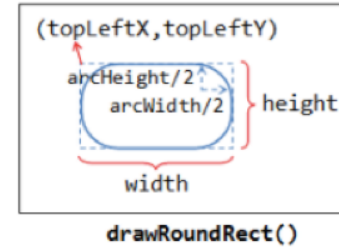
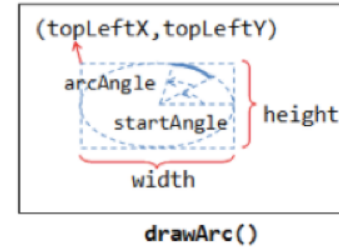
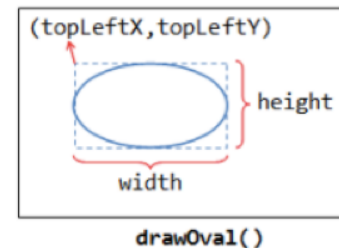
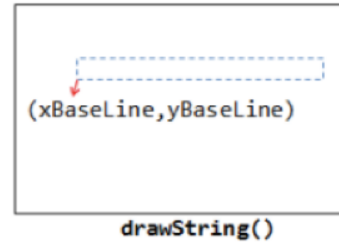
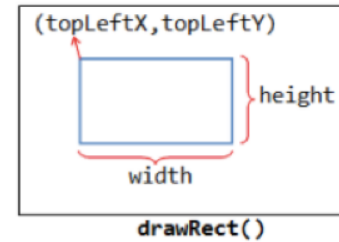
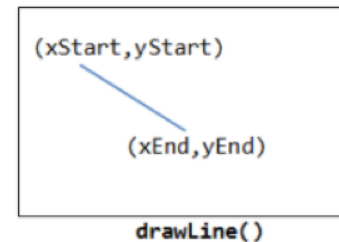
- ▶ Create an extension of component (JPanel or JFrame) and implement `paint` method in subclass.
- ▶ At start of `paint()` method cast `g` to `Graphics2D`.
- ▶ Call `repaint()` every time you want the component to be redrawn.

Geometric objects

- ▶ Objects from classes `Rectangle2D.Double`, `Line2D.Double`, etc. from `java.awt.geom`
- ▶ Constructors take parameters `x`, `y`, `width`, `height` but don't draw object.
- ▶ `Rectangle2D.Double`
- ▶ `Ellipse2D.Double`
- ▶ `Arc2D.Double`
- ▶ etc.

Drawing

- ▶ `myObj.setFrame(x, y, width, height)`: moves and sets size of component
- ▶ `g2.draw(myObj)`: gives outline
- ▶ `g2.fill(myObj)`: gives filled version
- ▶ `g2.drawString("a string", x, y)`: draws string



java.awt.Color



Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

Action listeners

- Define what should be done when a user performs certain operations.
 - e.g., clicks a button, chooses a menu item, presses Enter, etc.
- The application should implement the [ActionListener](#) interface.
- An instance of the application should be registered as a listener on one or more components.
- Implement the `actionPerformed` method.

```
public class MultiButtonApp implements ActionListener {  
    ...  
    //where initialization occurs:  
    button1.addActionListener(this);  
    button2.addActionListener(this);  
  
    ...  
    public void actionPerformed(ActionEvent e) {  
        if(e.getSource() == button1){  
            //do something  
        }  
    }  
}
```

Mouse listeners

- Define what should be done when a user enters a component, presses or releases one of the mouse buttons.
- The application should implement the [MouseListener](#) interface
 - Implement methods `mousePressed`, `mouseReleased`, `mouseEntered`, `mouseExited`, and `mouseClicked`.
- **Or** extend the [MouseAdapter](#) class
 - Which has default implementations of all of them.

```
public class MouseEventDemo ... implements MouseListener {
    //where initialization occurs:
    //Register for mouse events on blankArea and the panel.
    blankArea.addMouseListener(this);
    addMouseListener(this);
    ...

    public void mousePressed(MouseEvent e) {
        saySomething("Mouse pressed; # of clicks: "
            + e.getClickCount(), e);
    }
}
```

Lecture 4: The Catch-All Java Lecture

- ▶ Packages
- ▶ JavaDoc
- ▶ Exceptions
- ▶ Assertions
- ▶ Text I/O
- ▶ Java GUIs
- ▶ Graphics
- ▶ Events

Readings:

- ▶ Oracle's guides:
 - ▶ JavaDoc: <https://www.oracle.com/technetwork/articles/java/index-137868.html>
 - ▶ Exceptions: <https://docs.oracle.com/javase/tutorial/essential/exceptions/>
 - ▶ Assertions: <https://docs.oracle.com/javase/8/docs/technotes/guides/language/assert.html>
 - ▶ I/O: <https://docs.oracle.com/javase/tutorial/essential/io>
 - ▶ Writing Event Listeners: <https://docs.oracle.com/javase/tutorial/uiswing/events/index.html>
- ▶ Java Graphics: <https://github.com/pomonacs622021fa/Handouts/blob/master/graphics.md>
- ▶ Programming with GUIs: <https://github.com/pomonacs622021fa/Handouts/blob/main/JavaGUI.pdf>
- ▶ Swing/GUI Cheat Sheet: <https://github.com/pomonacs622021fa/Handouts/blob/master/swing.md>
- ▶ Textbook:
 - ▶ Chapter 1.2 (Page 107)