

# GC01 Introductory Programming

Week 4 – Lecture



Dr Ghita Kouadri Mostefaoui  
Department of Computer Science

# Outline

- Abstract classes
- Wrapper classes
- Exception handling
- Graphical user interfaces
  - Swing

# Abstract Classes

# Abstract Classes

- An *abstract class* is a placeholder in a class hierarchy that represents a generic concept
- An abstract class **cannot be instantiated**  
*we can also have abstract method*
- We use the modifier `abstract` on the class header to declare a class as abstract:

```
public abstract class Whatever
{
    // contents
}
```

# Why abstract classes?

- The use of abstract classes is a design decision – it helps us establish common elements in a class that is too general to instantiate
- Note that the `abstract` keyword is used to denote both an abstract method, and an abstract class.
- An abstract method has no definition, e.g.  
`makeNoise()`

```
public abstract class Animal
{
    public void eat(String food)
    {
        // do something with
        // food....
    }

    public void sleep(int hours)
    {
        // do something
    }
}
```

```
public abstract void makeNoise();
```

*It doesn't contain the behaviour,  
but it can have children that  
will define the behaviour.*

# Abstract methods must be overridden by subclasses

- The child of an abstract class *must override* the abstract methods of the parent, or it too will be considered abstract.

# Abstract methods must be overridden by subclasses

Now, any animal that wants to be instantiated (like a dog or cow) must implement the *makeNoise()* method - otherwise it is impossible to create an instance of that class.

```
public class Dog extends Animal
{
    public void makeNoise() {
        System.out.println ("Bark! Bark!");
    }
}
```

```
public class Cow extends Animal
{
    public void makeNoise() {
        System.out.println ("Moo! Moo!");
    }
}
```

# Rules

- An abstract class often contains abstract methods with no definitions (like an interface does)
- Unlike an interface, the `abstract` modifier must be applied to each abstract method
- An abstract class typically contains non-abstract methods (with bodies), further distinguishing abstract classes from interfaces
- An abstract method cannot be defined as
  - `final` (because it must be overridden) or
  - `static` (because it has no definition yet)
- A class declared as abstract does not need to contain abstract methods
  - We generally do that to prevent it from being instantiated.



# Wrapper Classes

# Why wrapper classes?

- Java's *primitive* data types (boolean, int, etc.) are not classes.
- Wrapper classes are used in situations where objects are required, such as for elements of a Collection (ArrayList, HashMap, etc)

```
ArrayList<Integer> aList=new  
                        ArrayList<Integer>();
```



The way we store data  
changes the way we search  
it, and the way we access it

# Primitives & Wrappers

- Java has a *wrapper* class for each of the eight primitive data types:
- Package *java.lang.\**

Primitive Type	Wrapper Class	Primitive Type	Wrapper Class
boolean	Boolean	float	Float
byte	Byte	int	Integer
char	Character	long	Long
double	Double	short	Short

# Value => Object

## Wrapper Object Creation

- *Wrapper.valueOf()* takes a value (or string) and returns an object of that class:

```
Integer i1 = Integer.valueOf(42);
```

```
Integer i2 = Integer.valueOf("42");
```

```
Boolean b1 = Boolean.valueOf(true);
```

```
Boolean b2 = Boolean.valueOf("true");
```

# Object => Value

## Unwrapping

- Each wrapper class Type has a method *typeValue* to obtain the object's value:

```
Integer i = Integer.valueOf(42); //wrapping
```

```
Boolean b = Boolean.valueOf("false"); //wrapping
```

```
System.out.println(i.intValue()); //unwrapping
```

```
System.out.println(b.booleanValue()); //unwrapping
```

### Output

42

false

# Exceptions

# Exceptions

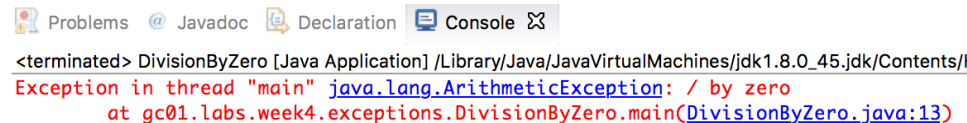
- *Exceptions* indicate problems that occur during a program's execution
- When an error occurs within a method, the method creates an *object* and hands it off to the runtime system
- The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred

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

        int a =0;

        System.out.println(3/a);

    }
}
```



The screenshot shows an IDE interface with tabs for Problems, Javadoc, Declaration, and Console. The Console tab is active, displaying the following text: <terminated> DivisionByZero [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0\_45.jdk/Contents/... Exception in thread "main" java.lang.ArithmeticException: / by zero at gc01.labs.week4.exceptions.DivisionByZero.main(DivisionByZero.java:13). The exception message and the file path are highlighted in red.

```
<terminated> DivisionByZero [Java Application] /Library/Java/JavaVirtualMachines/jdk1.8.0_45.jdk/Contents/...
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at gc01.labs.week4.exceptions.DivisionByZero.main(DivisionByZero.java:13)
```

# Exceptions Levels

Exceptions can occur at many levels

- Hardware/operating system level
  - Arithmetic exceptions; divide by 0.
  - Memory access violations; stack overflow.
- Language level
  - Type conversion; illegal values.
  - Bounds violations; illegal array indices.
  - Bad references; null pointers.
- Program level
  - User defined exceptions.



# Traditional Methods of Handling Errors

- In most procedural languages, the standard way of indicating an error condition is by returning an **error code**.
- The calling code typically did one of the following:
  - Testing the error code and taking the appropriate action.
  - Ignoring the error code.
- This often lead to very complex code.

# Traditional Methods of Handling Errors

## Pseudo-code

```
readFile {  
    open the file;  
    determine its size;  
    allocate that much memory;  
    read the file into memory;  
    close the file;  
}
```

At first glance, this function seems simple enough, but it ignores all the following potential errors.

1. What happens if the file can't be opened?
2. What happens if the length of the file can't be determined?
3. What happens if enough memory can't be allocated?
4. What happens if the read fails?
5. What happens if the file can't be closed?

```

readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}

```

# The Problem

- There's so much error detection, reporting, and returning here that the original seven lines of code are lost in the clutter. The logical flow of the code has also been lost, thus making it difficult to tell whether the code is doing the right thing:
- It's even more difficult to ensure that the code continues to do the right thing when you modify the method three months after writing it.
- Many programmers solve this problem by simply ignoring it — **errors are reported when their programs crash.**
- Makes the program *difficult to read, modify, maintain and debug*
- Remember: cost of code maintenance is very high

# Solution: Exception Handling

- If the `readFile` function used exceptions instead of traditional error-management techniques, it would look more like the following.
- Exceptions enable you to write the main flow of your code and to *deal with the exceptional cases elsewhere*
- Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you *organize the work more effectively*.
- Exception handling also helps grouping and differentiating Error types.

```
readFile {  
    try {  
        open the file;  
        determine its size;  
        allocate that much memory;  
        read the file into memory;  
        close the file;  
    } catch (fileOpenFailed) {  
        doSomething;  
    } catch (sizeDeterminationFailed) {  
        doSomething;  
    } catch (memoryAllocationFailed) {  
        doSomething;  
    } catch (readFailed) {  
        doSomething;  
    } catch (fileCloseFailed) {  
        doSomething;  
    }  
}
```

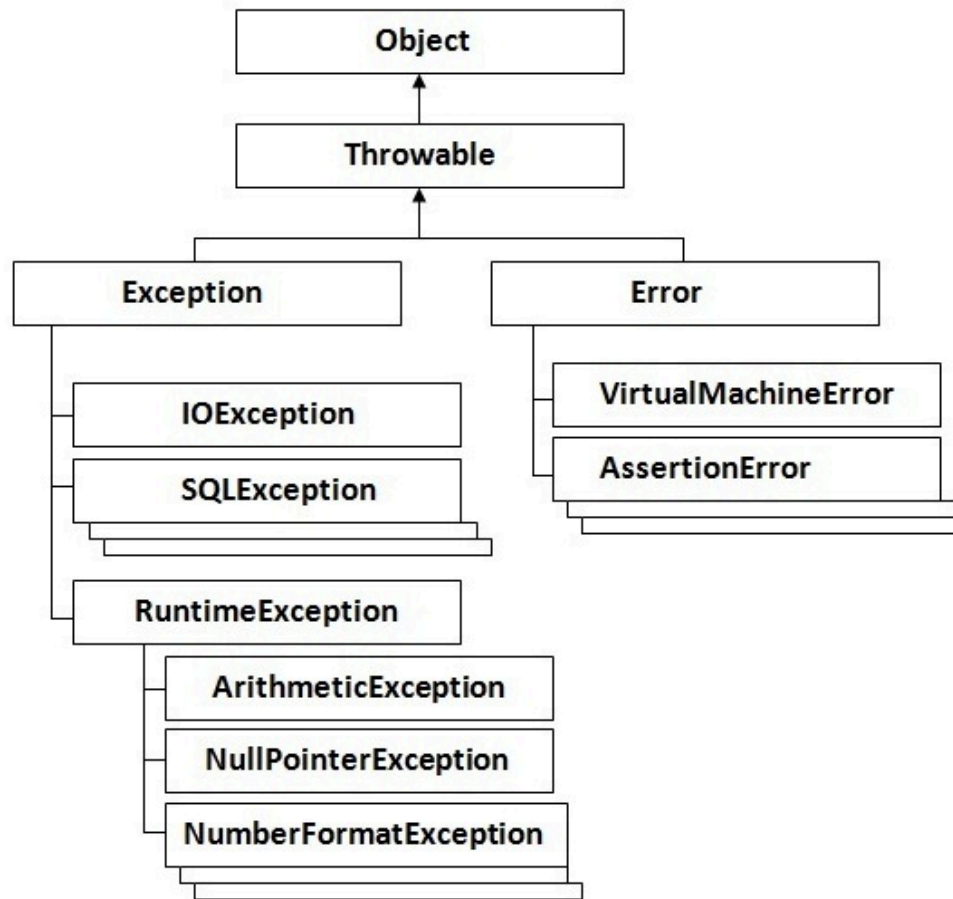
# Exception Handling - Advantages

- Can resolve exceptions
  - Allow a program to continue executing or
  - Notify the user of the problem and
  - Terminate the program in a *controlled manner*
- Exceptions in Java separates error handling from main business logic
- Based on ideas developed in Ada, Eiffel and C++
- Makes programs *robust* and *fault-tolerant*

# Demo

- [https://www.youtube.com/watch?v=gp\\_D8r-2hwk](https://www.youtube.com/watch?v=gp_D8r-2hwk)
- Dowson, M. (March 1997). "*The Ariane 5 Software Failure*". *Software Engineering Notes* 22 (2): 84. doi:10.1145/251880.251992.

# Exceptions Hierarchy





# Errors vs. Exceptions

- An *Error* is a subclass of `Throwable` that indicates serious problems that a reasonable application should not try to catch. Most such errors are abnormal conditions.
- *Error is irrecoverable* e.g. `OutOfMemoryError`, `VirtualMachineError`, `AssertionError` etc.
- Errors should not be caught or handled (except in the rarest of cases).

# Errors vs. Exceptions

- The class *Exception* and its subclasses are a form of `Throwable` that indicates conditions that a reasonable application might want to catch.
- *Exceptions are often recoverable* and even when not, they generally just mean an attempted operation failed, but your program can still carry on.

# Some of Java's important exceptions

Class	Description
ArithmeticException	Division by zero or some other arithmetic problem
ArrayIndexOutOfBoundsException	An array index is less than zero or greater than or equal to the array's length
FileNotFoundException	Reference to a file that cannot be found
IllegalArgumentException	Calling a method with an improper argument
StringIndexOutOfBoundsException	A String index is less than zero or greater than or equal to the String's length
NullPointerException	Reference to an object that has not been instantiated

# Examples

- If we divide any number by zero, there occurs an `ArithmeticException`.

```
int a=50/0;//ArithmeticException
```

- If we have null value in any object, performing any operation by the object occurs an `NullPointerException`.

```
String s=null;  
System.out.println(s.length()); //NullPointerException
```

- Suppose I have a string variable that have characters, converting this variable into digit will occur *NumberFormatException*.

```
String s="abc";  
int i=Integer.parseInt(s); //NumberFormatException
```

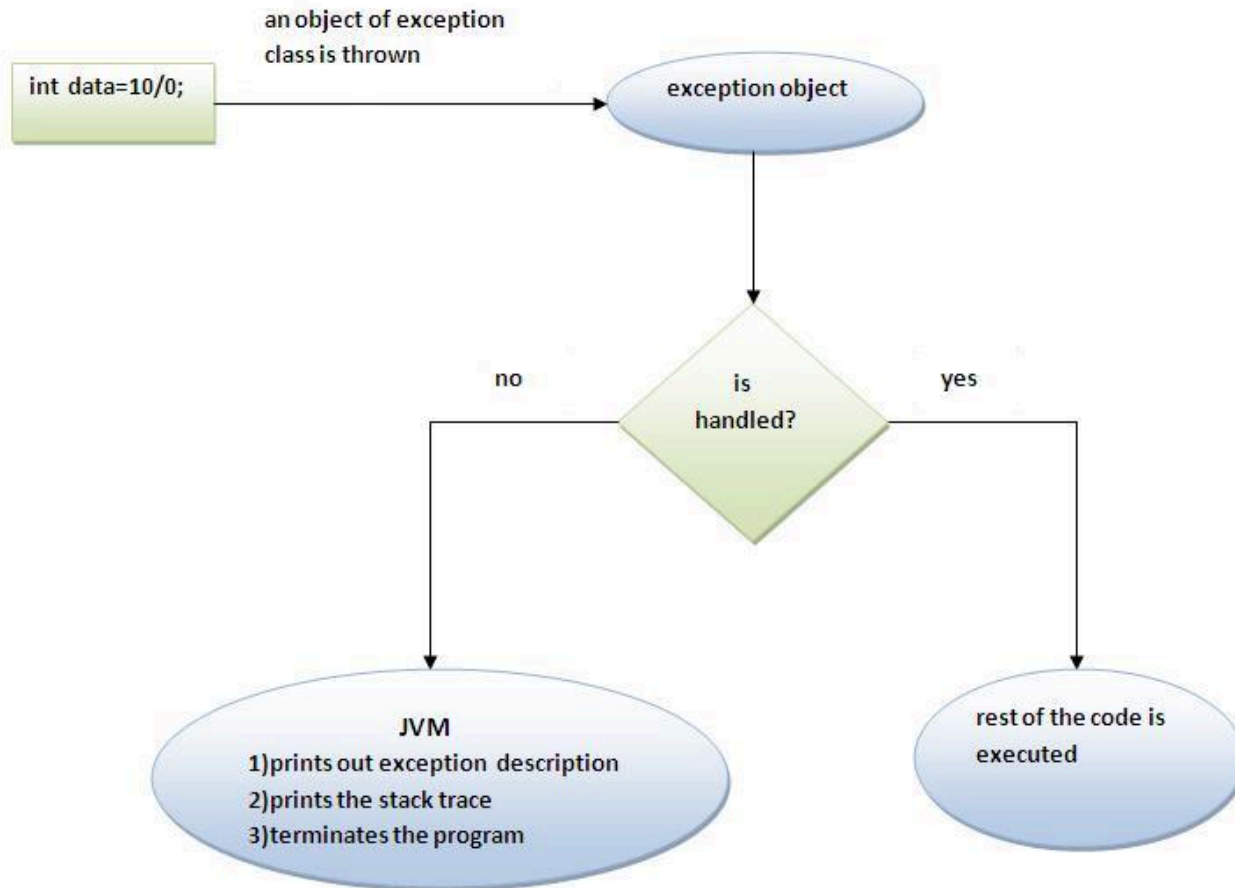
# Exceptions Handling - Syntax

```
try{
    // Code which might throw an exception
    //...
} catch (FileNotFoundException x) {
    // code to handle a FileNotFoundException exception
} catch (IOException x) {
    // code to handle any other I/O exceptions
} catch (Exception x) {
    // Code to catch any other type of exception
} finally{
    // This code is ALWAYS executed whether an exception was
    // thrown or not. A good place to put clean-up code.
    // close any open files, etc...
}
```

# try-catch Example

```
public class Testtrycatch{  
    public static void main(String args[]){  
        int a = 0;  
        try{  
            int data=10/a;  
        }catch(ArithmeticException e)  
            {System.out.println(e);}   
  
        System.out.println("rest of the code...");  
    }  
}
```

# Internal working of java try-catch block



# Java multi catch block

```
public class MultiCatch {  
    public static void main(String[] args) {  
        try{  
            int a[]=new int[5];  
            a[5]=30/0;  
        }  
        catch(ArithmeticException e){  
            System.out.println("task1 is completed");  
        }  
        catch(ArrayIndexOutOfBoundsException e){  
            System.out.println("task2 is completed");  
        }  
        catch(Exception e){  
            System.out.println("common task completed");  
        }  
  
        System.out.println("rest of the code...");  
    }  
}
```



# Rules

- Rule 1: At a time only one Exception is occurred and at a time only one catch block is executed.
- Rule 2: All catch blocks must be ordered from *most specific to most general* i.e. catch for `ArithmeticException` must come before catch for `Exception` . Otherwise, compilation error.

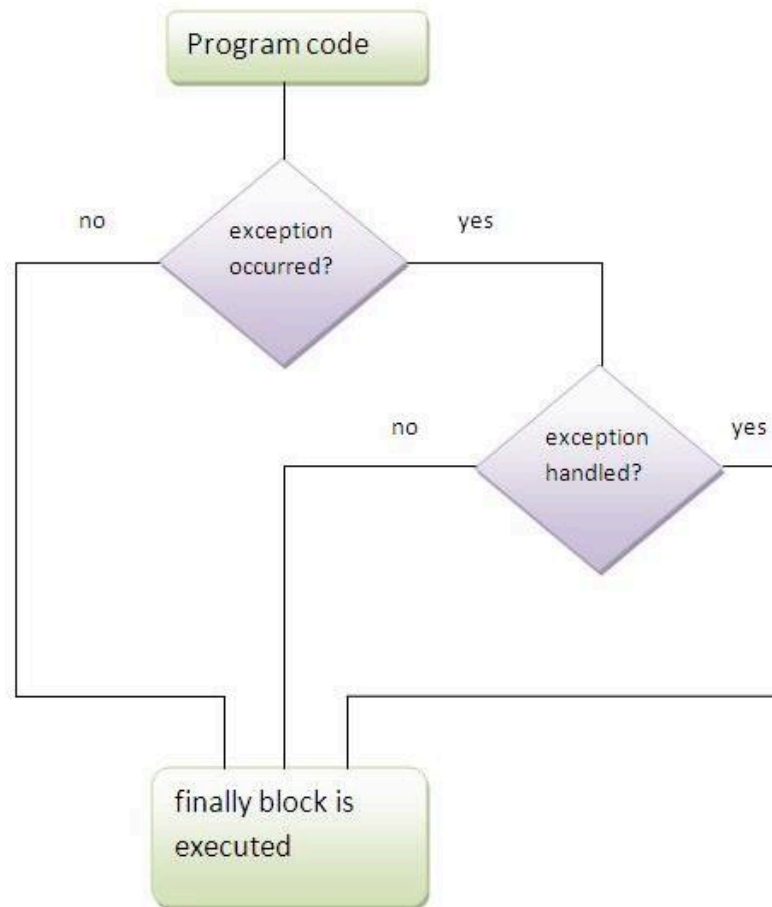
```
public class ExceptionOrdering {  
    public static void main(String args[]){  
        try{  
            int a[]=new int[5];  
            a[5]=30/0;  
        }  
        catch(Exception e){  
            System.out.println("common task completed");}  
        catch(ArithmeticException ex){  
            System.out.println("task1 is completed");}  
        catch(ArrayIndexOutOfBoundsException ef){  
            System.out.println("task 2 completed");}  
  
        System.out.println("rest of the code...");  
    }  
}
```

> Compile-time error

# Java finally block

- Java finally block is *always executed whether exception is handled or not.*
- Java Finally block in java can be used to put "cleanup" code such as closing a file, closing connection, etc.

# Internal working of java `finally` block



```
public class TestFinallyBlock{

    public static void main(String args[]){
        try{
            int data=25/0;
            System.out.println(data);
        }
        catch(ArithmeticException e){System.out.println(e);}
        finally{
            System.out.println("finally block is always
                               executed");
        }
        System.out.println("rest of the code...");
    }
}
```

# Rules

- Rule 1: The `try` statement should contain at least one `catch` block or a `finally` block and may have multiple catch blocks.
- Rule 2: The `finally` block is optional.
- Rule 3: The `finally` block will not be executed if program exits (either by calling `System.exit()` or by causing a fatal error that causes the process to abort).

# Java Custom Exceptions

- Java custom exceptions are used to customize the exception according to user need.
- By the help of custom exception, you can have your own exception and message.
- Let's see a simple example of java custom exception.

# Creating a custom exception

```
class InvalidAgeException extends Exception{  
  
    InvalidAgeException(String s){  
        super(s);  
    }  
}
```



# Using a custom exception

```
class TestCustomException {  
  
    static void validate(int age) throws InvalidAgeException {  
        if (age < 18)  
            throw new InvalidAgeException("age not valid");  
        else  
            System.out.println("welcome to vote");  
    }  
  
    public static void main(String args[]) {  
        try {  
            validate(13);  
        } catch (Exception m) {  
            System.out.println("Exception occurred: " + m);  
        }  
  
        System.out.println("rest of the code...");  
    }  
}
```

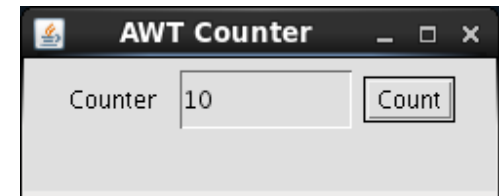
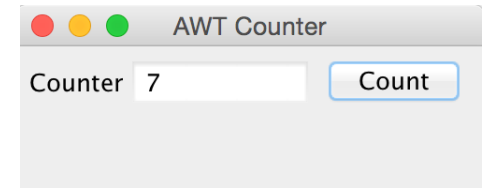
# Recap

- A program can catch exceptions by using a combination of the try, catch, and finally blocks.
- The try block identifies a block of code in which an exception can occur.
- The catch block identifies a block of code, known as an exception handler, that can handle a particular type of exception.
- The finally block identifies a block of code that is guaranteed to execute, and is the right place to close files, recover resources, and otherwise clean up after the code enclosed in the try block.
- The try statement should contain at least one catch block or a finally block and may have multiple catch blocks.

# Graphical User Interfaces (GUI)

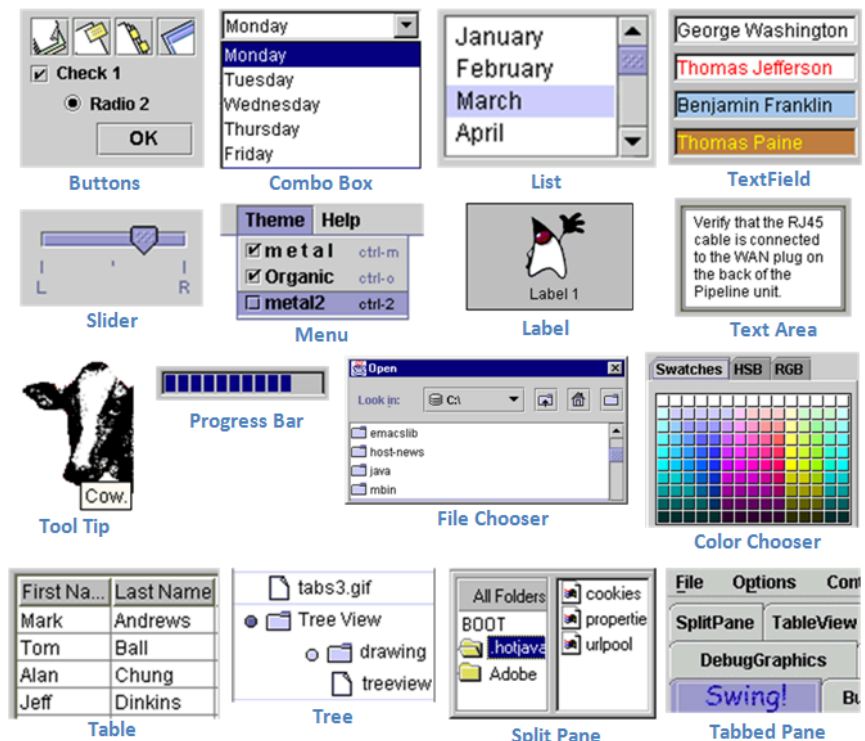
# Java AWT

- First release in 1995
- Java AWT (Abstract Windowing toolkit) is an API (Application Programming Interface) to develop GUI, or java window-based application in java
- Java AWT components are *platform-dependent* i.e. components are displayed according to the view of operating system.
- The *java.awt* package provides classes to build components such as TextField, Label, TextArea, RadioButton, CheckBox, Choice, List etc.



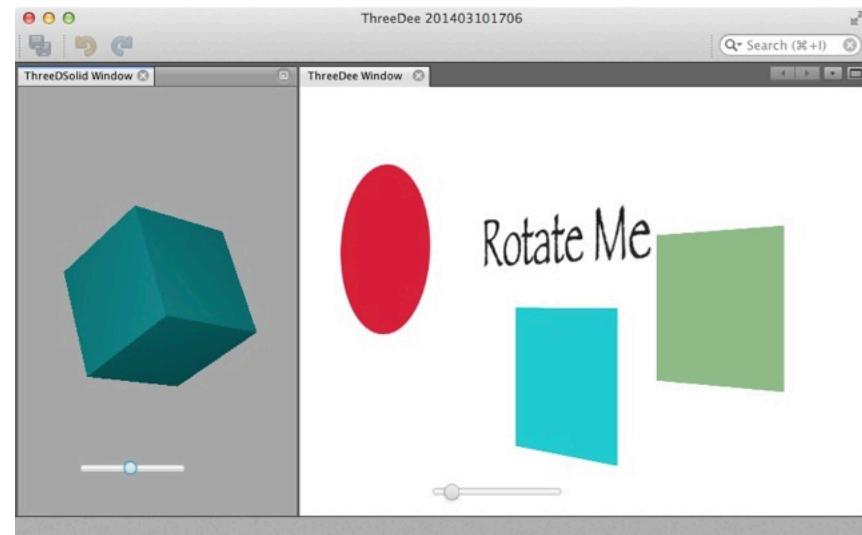
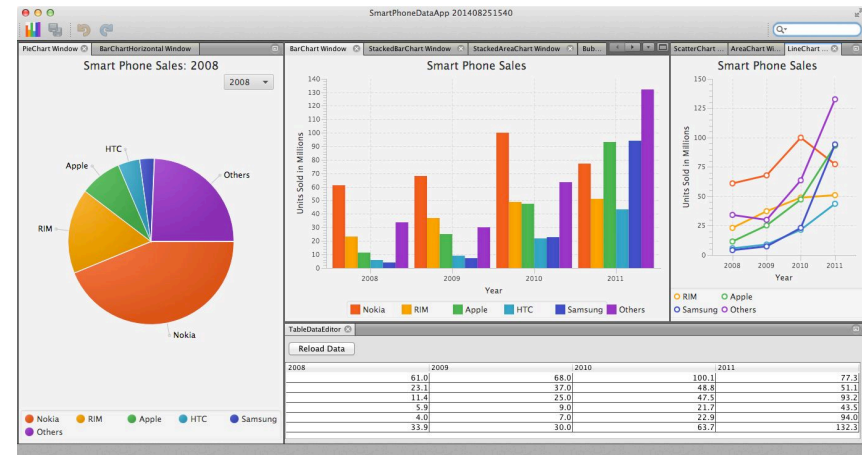
# Java Swing

- First release in 1996
- Swing is a GUI widget toolkit for Java. It is an API for providing a graphical user interface (GUI) for Java programs.
- Swing was developed to provide a more sophisticated set of GUI components than the earlier Abstract Window Toolkit (AWT).
- Swing provides a native look and feel that emulates the look and feel of several platforms, and also supports a pluggable look and feel that allows applications to have a look and feel unrelated to the underlying platform.



# JavaFX

- First release in 2011
- *JavaFX* is a software platform for creating and delivering desktop applications, as well as rich internet applications (RIAs) that can run across a wide variety of devices.
- Charts, 3D API, etc
- *JavaFX* is intended to replace Swing as the standard GUI library for Java SE, but *both will be included for the foreseeable future*.



Swing

# Three Types of GUI Classes

## **1. Containers**

JFrame, JPanel, JWindow

## **2. Components**

JButton, JTextField, JComboBox, JList, etc.

## **3. Helpers**

Graphics, Color, Font, Dimension, etc.



# Swing containers

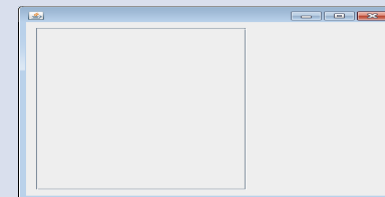
A JWindow object is a top-level window with no borders and no menubar.



A JFrame is a top-level window with a title and a border



JPanel is the simplest container. It provides space in which any other component can be placed, including other panels.

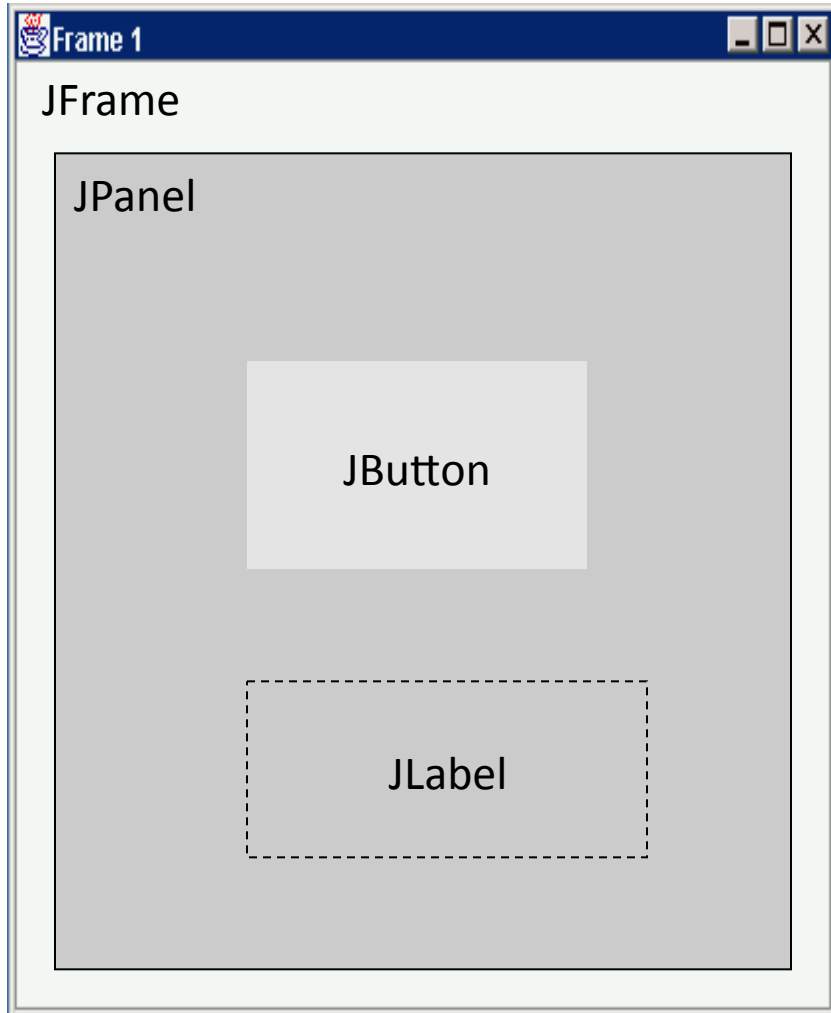


# Swing Components

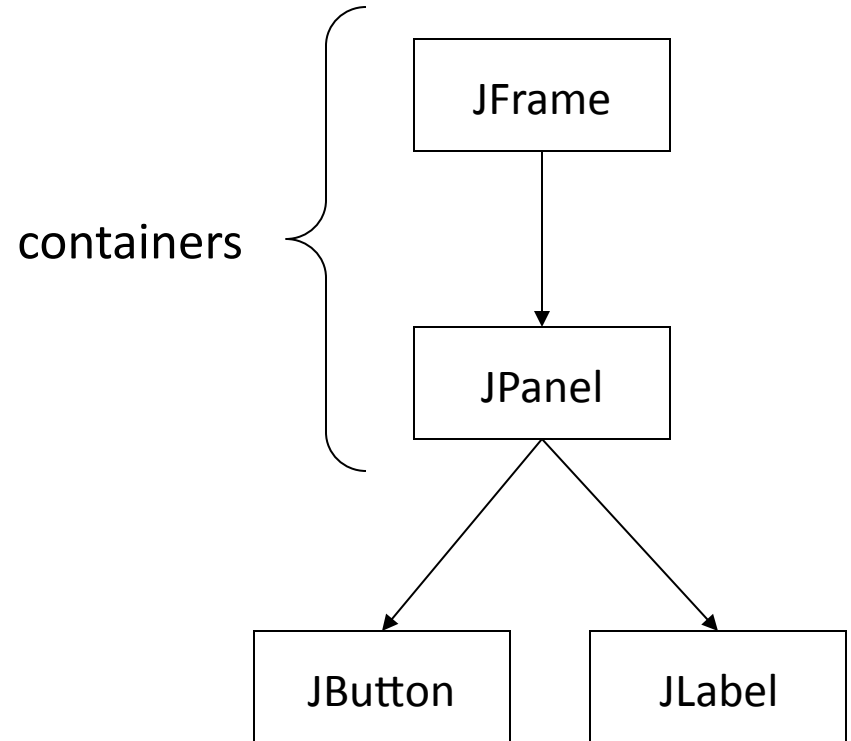
- JComponent
  - JColorChooser, JComboBox, JFileChooser, JInternalFrame, JLabel, JLayer, JLayeredPane, JList, JMenuBar, JOptionPane, JPanel, JPopupMenu, JProgressBar, JRootPane, JScrollbar, JScrollPane, JSeparator, JSlider, JSpinner, JSplitPane, JTabbedPane, JTable, JTableHeader, JTextComponent, JToolBar, JToolTip, JTree, ...
  - See Java API specification: *javax.swing.JComponent*  
<https://docs.oracle.com/javase/8/docs/api/>

# Anatomy of an Application GUI

## GUI

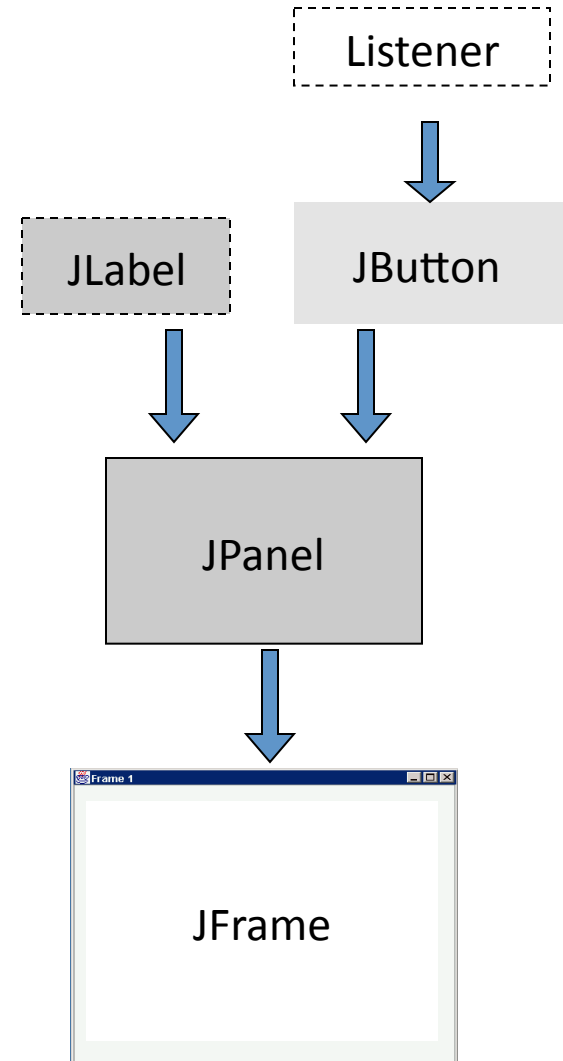


## Internal structure



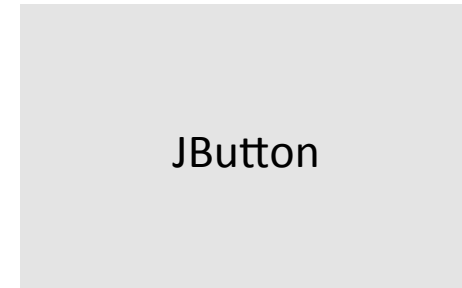
# Build From Bottom Up

- Create:
  - Frame
  - Panel
  - Components
  - Listeners
- Add: (bottom up)
  - listeners into components
  - components into panel
  - panel into frame



# GUI Component API

- Java: GUI component = class
- Properties
  - text, icon
- Methods
  - setText()
- Events
  - click



# Adding a GUI Component (2)

## 1. Create it

Instantiate object:

```
b = new JButton();
```

## 2. Configure it

Constructors: `new JButton("press me");`

Properties: `b.setText("press me");`

Methods: `b.setText("press me");`



press me

## 3. Add it to a parent container

```
panel.add(b);
```

## 4. Listen to it

Events: Listeners

# Application Code

```
class HelloGUI extends JFrame{
    JPanel p;
    JButton b;
    JLabel l;

    HelloGUI(){
        p = new JPanel();
        b = new JButton("press me");
        l = new JLabel("Hello!");

        p.add(b); // add button to panel
        p.add(l); // add label to panel

        add(p); // add panel to parent frame
    }

    public static void main(String[] args){

        HelloGUI hg = new HelloGUI();

        hg.setSize(200,200);
        hg.setVisible(true);
        //f.show(); //deprecated
    }
}
```

# Event Handling

- All components can listen for one or more *events*.
- Typical examples are:
  - Mouse movements
  - Mouse clicks
  - Hitting any key
  - Hitting return key
  - etc.
- Telling the GUI what to do when a particular event occurs is the role of the event handler.



# Event Handlers

- *Action listeners* are probably the easiest — and most common — event handlers to implement. You implement an action listener to define what should be done when an user performs certain operation.
- *An action event occurs*, whenever an action is performed by the user. Examples: When the user clicks a button, chooses a menu item, presses Enter in a text field. The result is that an *actionPerformed* message is sent to all action listeners that are registered on the relevant component.

# ActionEvent

- In Java, most components have a special event called an *ActionEvent*.
- This is loosely speaking the *most common* or *canonical event* for that component.
- A good example is a click for a button.
- To have any component listen for ActionEvents, you must register the component with an ActionListener. e.g.
  - `button.addActionListener(new MyAL());`

# Delegation Model

- When you register an *ActionListener* with a component, you must pass it the class which will handle the event – that is, do the work when the event is triggered.
- For an *ActionEvent*, this class must implement the *ActionListener* interface.
- This is a simple way of guaranteeing that the *actionPerformed* method is defined.

# actionPerformed

- The actionPerformed method has the following signature:  
    void actionPerformed(ActionEvent)
- The object of type *ActionEvent* passed to the event handler is used to query information about the event.
- Some common methods are:
  - getSource()
    - object reference to component generating event
  - getActionCommand()
    - some text associated with event (text on button, etc).
  - These methods are particularly useful when using one event handler for multiple components.

# Writing an Action Listener

To write an Action Listener, follow the steps given below:

1. Declare an event handler class and specify that the class either implements an *ActionListener interface* or extends a class that implements an ActionListener interface. For example:

```
public class MyClass implements ActionListener {
```

2. Register an instance of the event handler class as a listener on one or more components. For example:

```
someComponent.addActionListener(instanceOfMyClass);
```

3. Include code that implements the methods in listener interface. For example:

```
public void actionPerformed(ActionEvent e) {  
    ...//code that reacts to the action...  
}
```

# Application code with listeners

```
class HelloGUI extends JFrame implements ActionListener{
    JPanel p;
    JButton b;
    JLabel l;

    HelloGUI(){
        p = new JPanel();
        b = new JButton("press me");
        l = new JLabel("Hello!");

        b.addActionListener(this);

        p.add(b); // add button to panel
        p.add(l); // add label to panel

        add(p); // add panel to parent frame
    }

    public static void main(String[] args){
        ...
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked");
    }
}
```

# Notes

- Trail: Creating a GUI With JFC/Swing
  - <http://docs.oracle.com/javase/tutorial/uiswing/TOC.html>
- Hard coding a GUI is a legacy way of looking at user interface design.
- Swing GUI Designer (WindowBuilder)
  - <http://www.vogella.com/tutorials/EclipseWindowBuilder/article.html>