presentation slides for

# JAVA, JAVA, JAVA

## Object-Oriented Problem Solving
### Third Edition

Ralph Morelli | Ralph Walde
Trinity College
Hartford, CT

published by Prentice Hall

# Java, Java, Java

## Object Oriented Problem Solving

# Chapter 10: Exceptions

# Objectives

- Understand Java's exception handling mechanisms.

- Be able to use the Java try/catch statement.

- Know how to design effective exception handlers.

- Appreciate the importance that exception handling plays in program design.

- Be able to design your own Exception subclasses.

# Outline

- Introduction

- Handling Exceptional Conditions

- Java's Exception Hierarchy

- Handling Exceptions within a Program

- Error Handling and Robust Program Design

- Creating and Throwing Your Own Exceptions

- From the Java Library: JOptionPane

# Introduction

- No matter how well designed a program is, there is always the chance that some kind of error will arise during its execution.

- A well-designed program should include code to handle errors and other exceptional conditions when they arise.

- This chapter describes Java's exception handling features.

# Handling Exceptional Conditions

- The avgFirstN() method expects that $N > 0$.

- If $N = 0$, a *divide-by-zero* error occurs in *avg/N*.

```
/**
 * Precondition:  N > 0
 * Postcondition: avgFirstN() equals the average of (1+2+…+N)
 */
public double avgFirstN(int N) {
    double sum = 0;
    for (int k = 1; k <= N; k++)
        sum += k;
    return sum/N;        // What if N is 0 ??
} // avgFirstN()
```

Bad Design: Doesn't guard against divide-by-0.

# Traditional Error Handling

- Error-handling code built right into the algorithm:

```java
/**
  * Precondition:  N > 0
  * Postcondition: avgFirstN() equals the average of (1+2+…+N)
  */
public double avgFirstN(int N) {
    double sum = 0;
    if (N <= 0) {
        System.out.println("ERROR avgFirstN: N <= 0. Program terminating.");
        System.exit(0);
    }
    for (int k = 1; k <= N; k++)
        sum += k;
    return sum/N;       // What if N is 0 ??
} // avgFirstN()
```

It's sometimes risky to exit a program like this.

# Java's Exception Hierarchy

- *Unchecked exceptions* belong to a subclass of RuntimeException and are not monitored by the compiler.
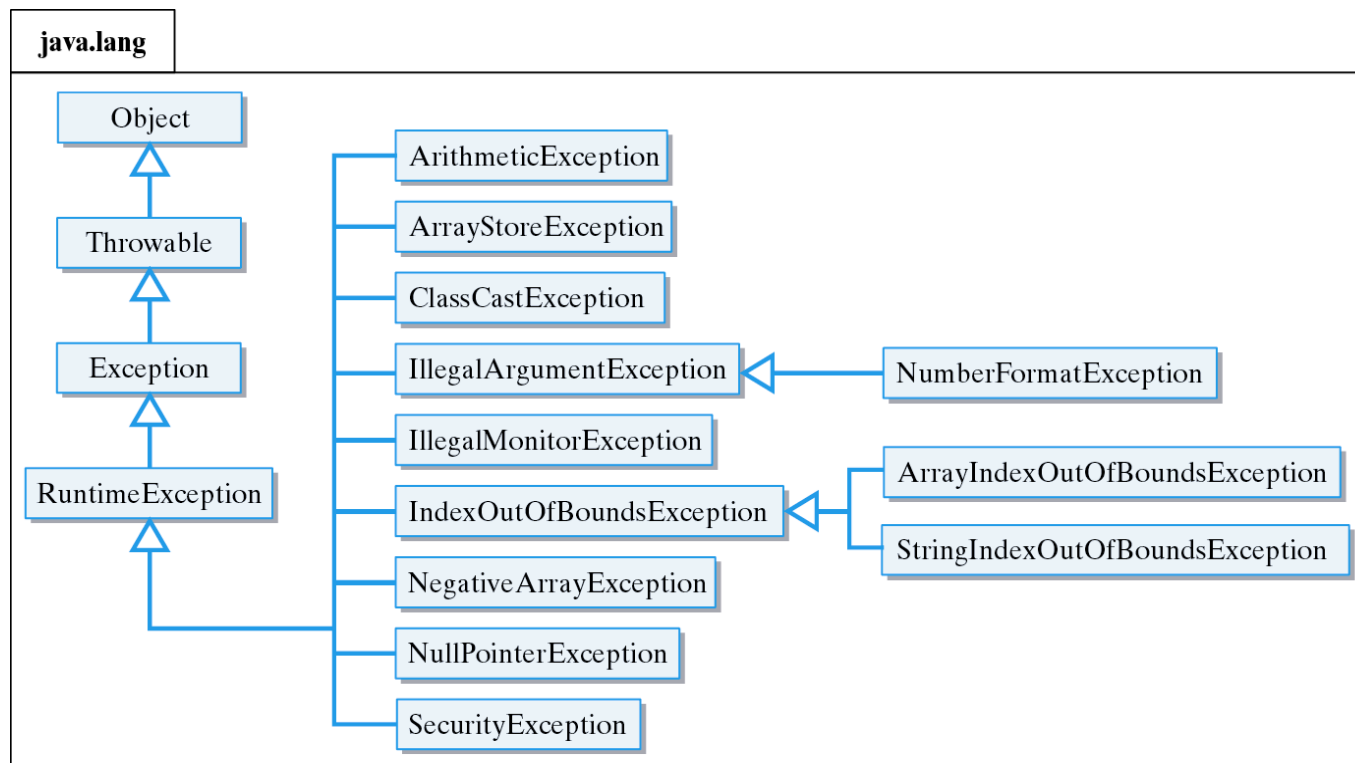
# Some Important Exceptions

| Class | Description |
|---|---|
| ArithmeticException | Division by zero or some other kind of arithmetic problem |
| ArrayIndexOutOfBounds-Exception | An array index is less than zero or greater than or equal to the array's length |
| FileNotFoundException | Reference to a unfound file |
| IllegalArgumentException | Method call with improper argument |
| IndexOutOfBoundsException | An array or string index out of bounds |
| NullPointerException | Reference to an object which has not been instantiated |
| NumberFormatException | Use of an illegal number format, such as when calling a method |
| StringIndexOutOfBoundsException | A String index less than zero or greater than or equal to the String's length |

# Some Common Exceptions

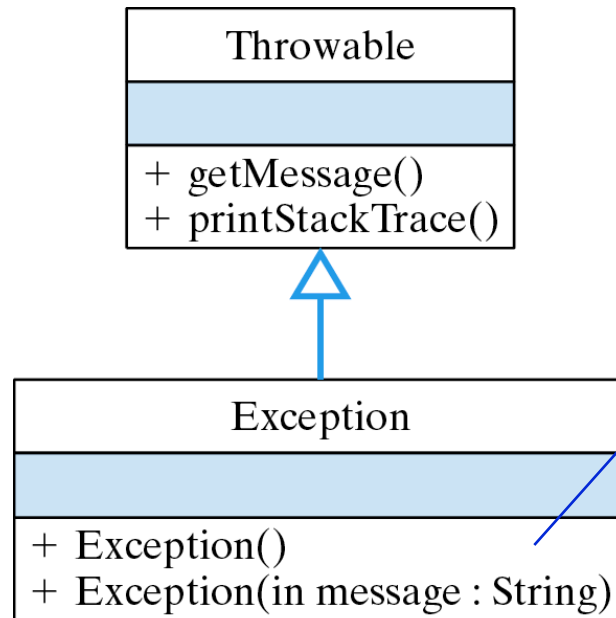| Class | Method | Exception Raised | Description |
|---|---|---|---|
| Double | valueOf(String) | NumberFormatException | The String is not a double |
| Integer | parseInt(String) | NumberFormatException | The String is not a int |
| String | String(String) | NullPointerException | The String is null |
| | indexOf(String) | NullPointerException | The String is null |
| | lastIndexOf(String) | NullPointerException | The String is null |
| | charAt(int) | StringIndexOutOfBounds Exception | The int is invalid index |
| | substring(int) | StringIndexOutOfBounds Exception | The int is invalid index |
| | substring(int,int) | StringIndexOutOfBounds Exception | An int is invalid index |

# Checked Exceptions

- *Checked exception* *must* either be caught or declared within the method where it is thrown.

- Monitored by the Java compiler.

- Example: IOException

**IOException must be declared ...**

```
public static void main(String argv[]) throws IOException {

    BufferedReader input = new BufferedReader
            (new InputStreamReader(System.in));

    String inputString = input.readLine();    // May throw IOException
}
```

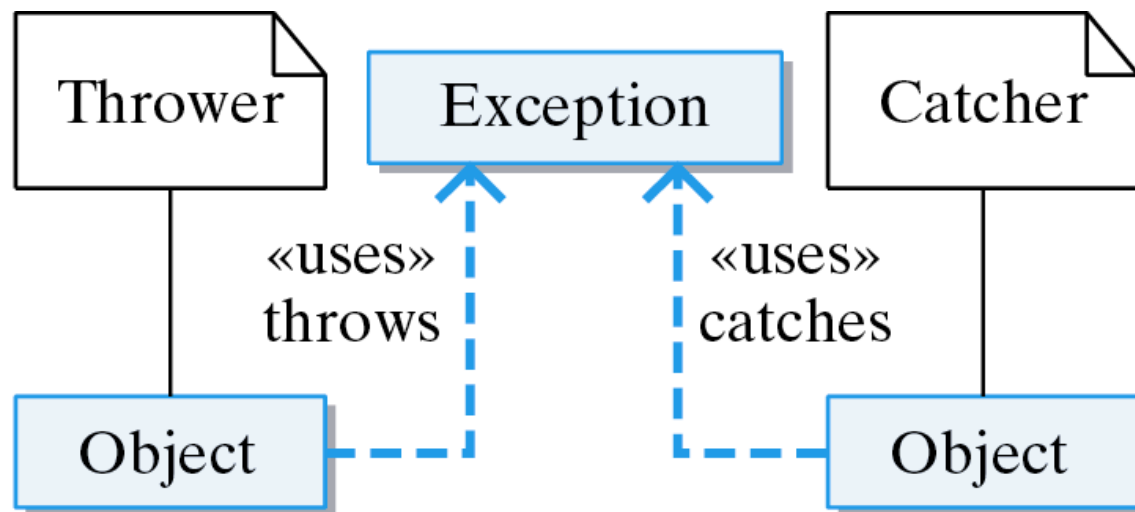**...because readLine() may cause it.**

# The Exception Class

Throwable
|  |
| --- |
| + getMessage()<br>+ printStackTrace() |

Exception
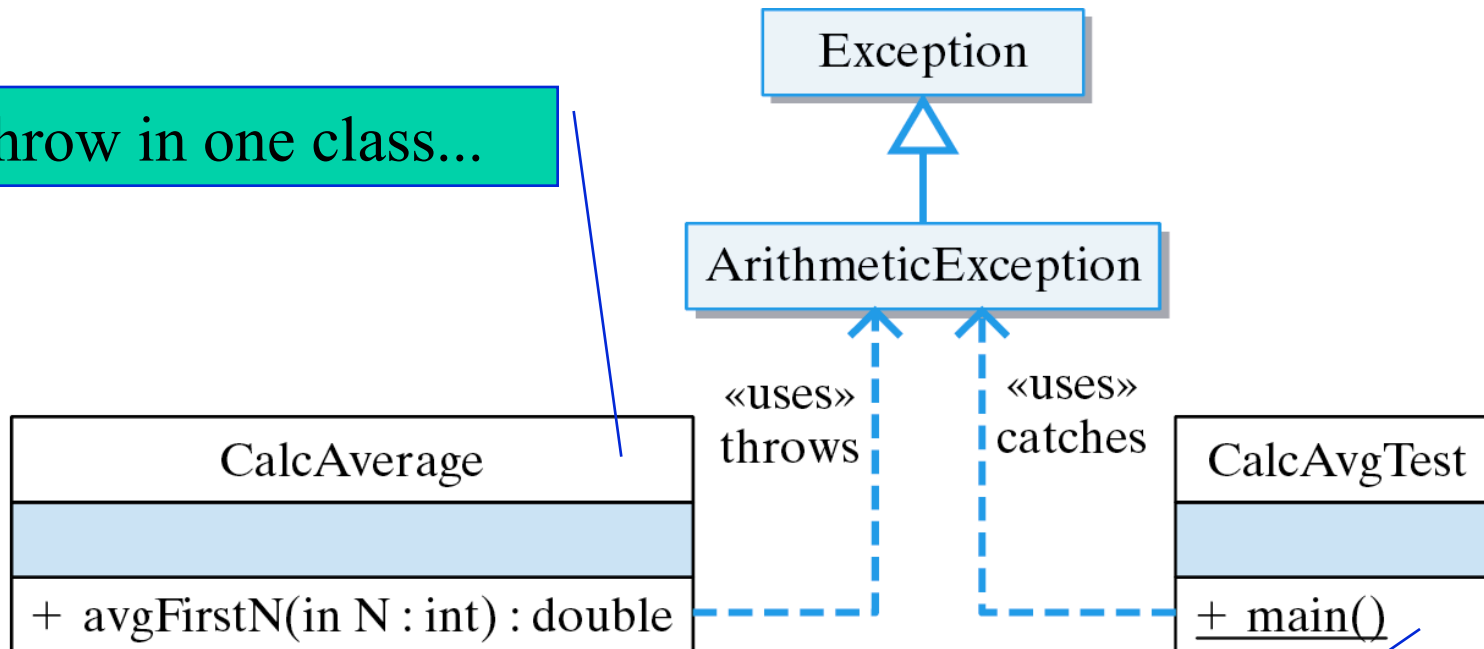|  |
| --- |
| + Exception()<br>+ Exception(in message : String) |

Simple: only constructor methods.

# Exception Handling

- When an exception occurs, an object will throw an exception. The *exception handler,* possibly the same object, will catch it.

# Example: Two Classes

Exception

Throw in one class...

ArithmeticException

«uses»
throws

«uses»
catches

CalcAverage

+ avgFirstN(in N : int) : double

CalcAvgTest

+ main()

…Catch in the other.

# Try, Throw and Catch

```java
public class CalcAverage {
  public double avgFirstN(int N ){
    double sum = 0;
    try {                                   // Try block: exception thrower
      if (N <= 0)
        throw new Exception("ERROR: Can't average 0 slements");
      for (int k = 1; k <= N; k++)
        sum += k;
      return sum/N;
    }
  }// avgFirstN()
} // CalcAverage

public class CalcAvgTest {
  public static void main(String args[]) {
    try {
      CalcAverage ca = new CalcAverage();
      System.out.println("AVG + " + ca.avgFirstN(0));
    } catch (ArithmeticException e) { // C
      System.out.println(e.getMessage());
      e.printStackTrace();
      System.exit(0);
    }
  }// main()
} // CalcAvgTest
```

Throw in one class...

...Catch in the other.

**Effective Design:** Java's exception handling mechanism allows you to separate normal code from exception handling code.

# Try/Throw/Catch

- A *try block* contains statements that may cause an exception. It signals your intention to handle the exception.

- Throwing an exception is like pulling the fire alarm. Once an exception is thrown, control is transferred to an appropriate catch clause.

- Exceptions are handled in the *catch clause*.

- The *finally block* is optional. Unless the program is exited, it is executed whether an exception is thrown or not.

# Try/Throw/Catch (cont)

```java
try {
        // Block of statements
        // At least one of which may throw an exception

        if ( /* Some condition obtains */ )
            throw new ExceptionName();

    } catch (ExceptionName ParameterName) {
        // Block of statements to be executed
        // If the ExceptionName exception is thrown in try
    }
  ...  // Possibly other catch clauses
    }   catch (ExceptionName2 ParameterName) {
        // Block of statements to be executed
        // If the ExceptionName2 exception is thrown in try
    } finally {
        // Optional block of statements that is executed
        // Whether an exception is thrown or not
}
```

# Restrictions on try/catch/finally

- A try block must be followed by one or more catch clauses.

- A catch clause may only follow a try block.

- A throw statement is used to throw both *checked* and *unchecked* exceptions.

- Unchecked exceptions belong to RuntimeException or its subclasses.

- Checked exceptions must be caught or declared.

- A throw statement must be contained within the *dynamic scope* of a try block, and the type of Exception thrown must match at least one of the try block's catch clauses. Or, the throw statement must be contained within a method or constructor that has a throws clause for the type of thrown Exception.

# Dynamic versus Static Scope

```
class MyClass{

    public void method1 () {
        int X = 1;
        System.out.println("Hello" + X);
    }

    public void method2 () {
        int Y = 2;
        System.out.println("Hello" + Y);
    }

    public static void main(String argv[]) {
        MyClass myclass = new MyClass();
        if(Math.random() > 0.5)
            myclass.method2 () ;
        else
            myclass.method1 () ;
    }

}
```
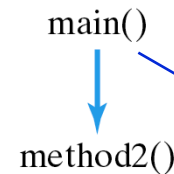
Static Scope: Follow the definitions.
Neither method1() nor method2() is
in the static scope of main().

```
                 MyClass
              /     |      \
        method1 () method2 ()  main ()
          |          |
          X          Y
```
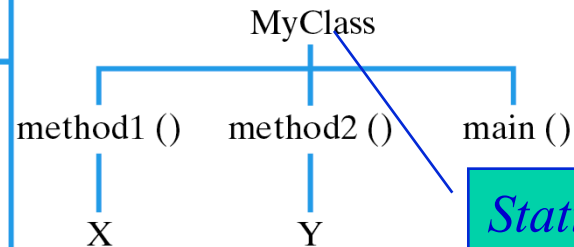
*Static scope*: how the program is written.

Dynamic Scope: Follow the execution.
If Math.random() > 0.5, method2()
is in the dynamic scope of main().

```
             main()
               |
               ↓
           method2()
```

*Dynamic scope:* how the program is run.

# The Method Call Stack

- The *method call stack*  keeps track of the methods that are called during program execution. The current method is on the top of the stack.

```java
public class Propagate{
  public void method1 (int n) {
    method2(n);
  }
  public void method2 (int n) {
    method3(n);
  }
  public void method3 (int n) {
    for(int k = 0; k < 5; k ++ ) {//Block1
      if(k % 2 == 0) {            //Block2
        System.out.println(k/n) ;
      }
    }
  }
  public static void main(String args[]) {
    Propagate p = new propagate() ;
    p.method1(0) ;
  }
}
```

Method Call Stack

The state of the stack on the first iteration of the for loop in method3().

| method3 () |
| n = 0 k = 0 |
| method2 () |
| n = 0 |
| method1 () |
| n = 0 |
| main() |
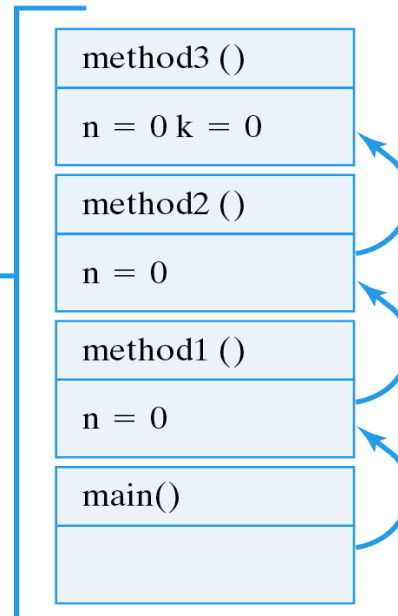
*Dynamic scope:*
main() calls method1() which calls method2() which calls method3().
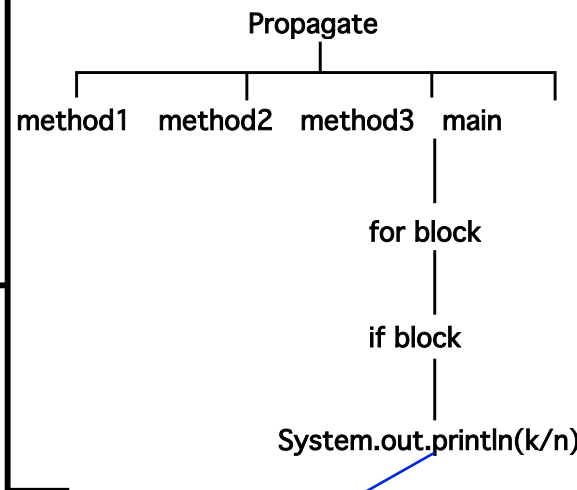
# Finding a Catch Block

- Search upward through the static scope and backward through the dynamic scope.

```
public class Propagate {
  public void method1(int n ) {
    method2(n);
  }
  public void method2(int n ) {
    method3(n);
  }
  public void method3(int n) {
    for (int k = 0; k < 5; k++ ) {     // Block1
      if ( k % 2 == 0 ) {              // Block2
        System.out.println( k / n );
      }
    }
  }
  public static void main(String args[]) {
    Propagate p = new Propagate();
    p.method1(0);
  }
}
```

**Static Scope Hierarchy**

Propagate

method1   method2   method3   main

for block

if block

System.out.println(k/n)

*Dynamic scope:* If no handler found in method3() Java searches method2(), then method1(), then main().

*Static scope:* If an error occurs in *k/n,* Java searches up this branch for an exception handler.

# Default Exception Handling

- Java can handle unchecked exceptions itself.

```java
public class CalcAverage {
  public double avgFirstN(int N ){
    double sum = 0;
    try {                              // Try block: exception thrower
      if (N <= 0)
         throw new Exception("ERROR: Can't average 0 slements");
      for (int k = 1; k <= N; k++)
          sum += k;
      return sum/N;
    }
  } // avgFirstN()
  public static void main(String args[]) {
      CalcAverage ca = new CalcAverage();
      System.out.println( "AVG + " + ca.avgFirstN(0));
  } // main()
} // CalcAverage
```

No catch clause for ArithmeticException, so Java handles the exception itself.

```
java.lang.ArithmeticException: ERROR: Can't average 0 elements
      at CalcAverage.avgFirstN(CalcAverage.java:9)
      at CalcAverage.main(CalcAverage.java:20)
      at com.mw.Exec.run(JavaAppRunner.java:47)
```

# Robust Program Design

- Four ways to handle exceptions:

| Kind of Exception | Kind of Program | Action to be Taken |
|---|---|---|
| Caught by Java | | Let Java handle it |
| Fixable condition | | Fix the error and resume execution |
| Unfixable condition | Stoppable | Report error and terminate |
| Unfixable condition | Not stoppable | Report error and resume processing |

- Your own programs: letting Java handle exceptions may be the best choice.

- During program development: exceptions help you identify bugs.

- Commercial software: the program should handle its exceptions because the user can't.

# Handling Strategies

- **Print Error Message and Terminate.** Unless the error can be fixed, it's better to terminate a program than to allow it to spread bad data -- e.g., the divide-by-zero example.

- **Log the Error and Resume**: A heart monitor program cannot be terminated.

# Erroneous Input Error

- Erroneous input should be anticipated and fixed.
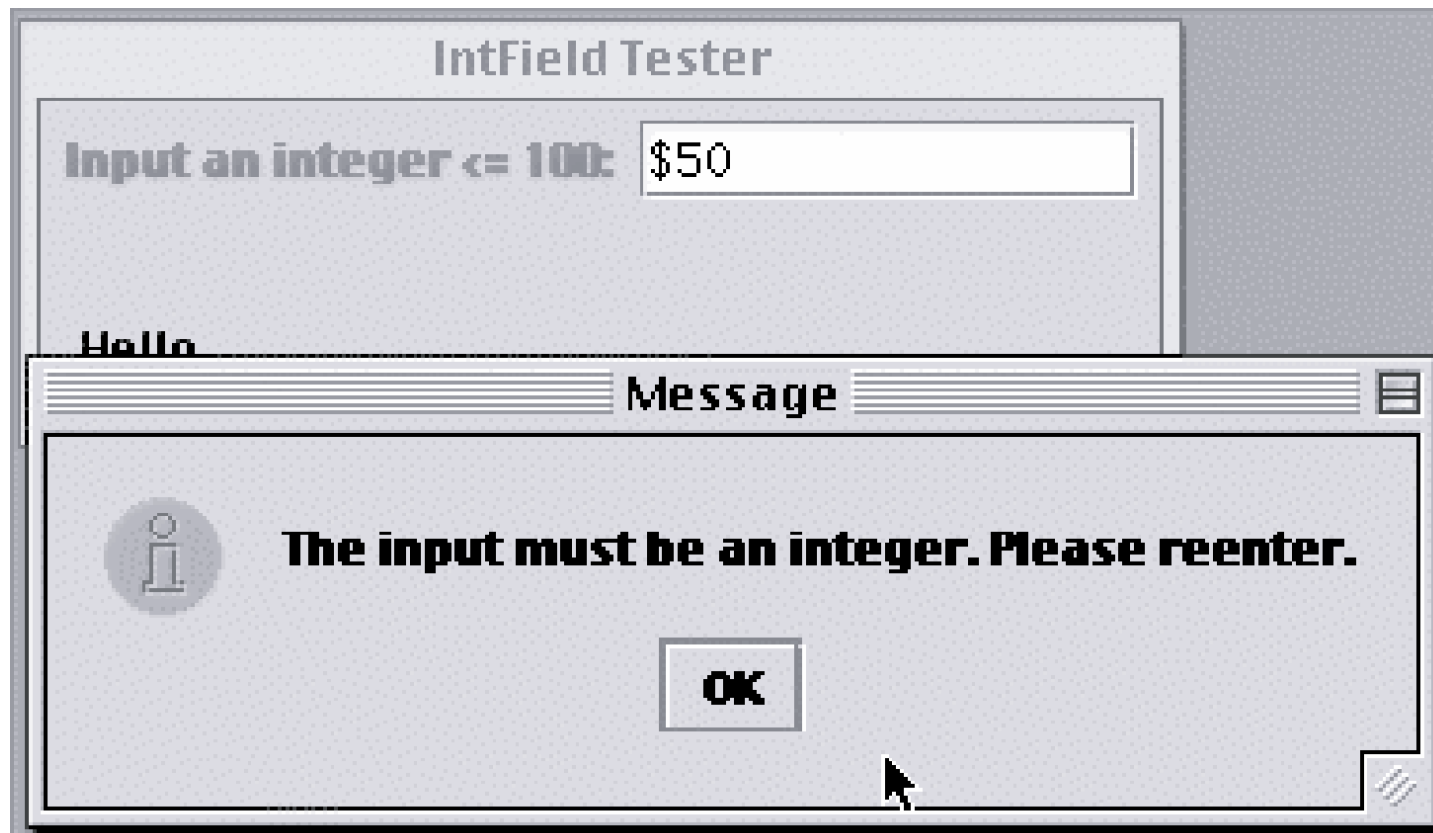
# Fixable Error: The IntField Class

- An IntField is a JTextField that accepts only integer input or throws an exception.

```
           ┌─────────────┐
           │  JTextField │
           └─────────────┘
                  △
                  │
                  │
    ┌───────────────────────────┐
    │         IntField          │
    ├───────────────────────────┤
    │                           │
    ├───────────────────────────┤
    │ + IntField()              │
    │ + IntField(in size : int) │
    │ + getInt() : int          │
    └───────────────────────────┘
```
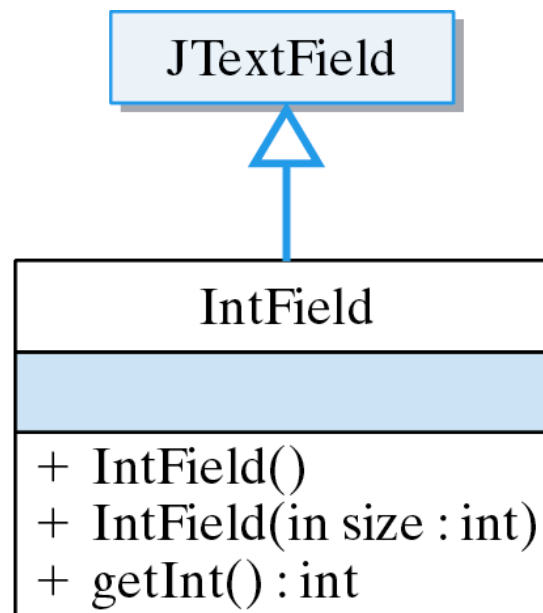
# The NumberFormatException

- If the user types a non-integer, the IntField throws a NumberFormatException, which is caught by the applet.
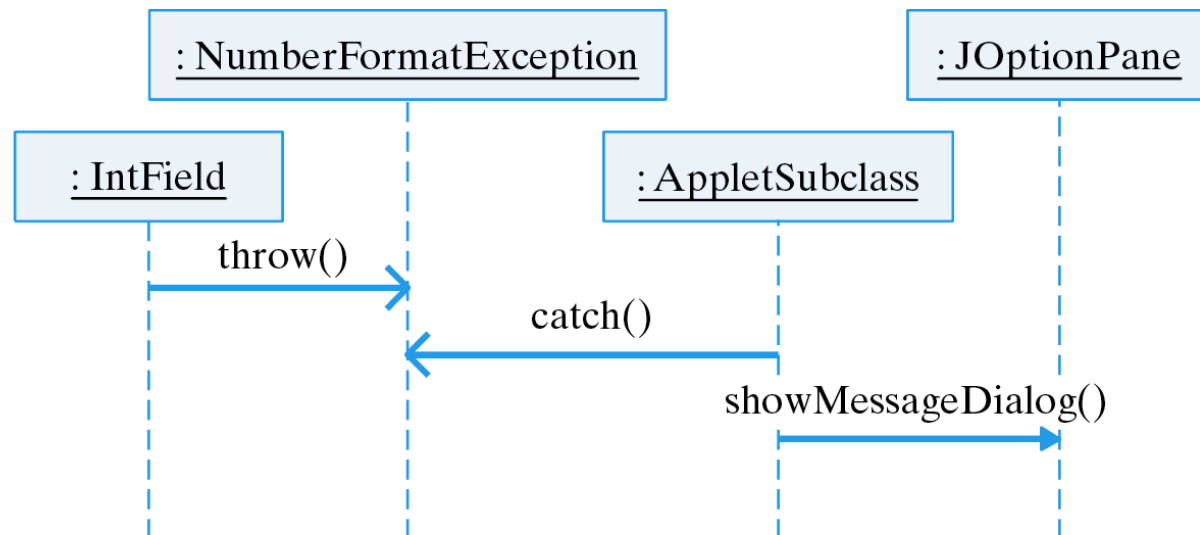
# Fix the Error and Resume

- Fixable Errors. If the user does not input an integer ...

```java
import javax.swing.*;

public class IntField extends JTextField {

    public IntField () {
        super();
    }
    public IntField (int size) {
        super(size);
    }

    public int getInt() throws NumberFormatException {
        return Integer.parseInt(getText());
    } // getInt()

} // IntField
```

parseInt() throws an exception when "57%" is input.

# Fix the Error and Resume (cont)

- …Print an error message and ask the user to reenter the data.

getInt() may throw an exception

```java
public void actionPerformed(ActionEvent e) {
    try {
        userInt = intField.getInt();
        message = "You input " + userInt + " Thank you.";
    } catch (NumberFormatException ex) {
        JOptionPane.showMessageDialog(this,
            "The input must be an integer. Please reenter.");
    } finally {
        repaint();
    }
} // actionPerformed()
```

# Example: To Fix or Not to Fix

- Should you use an exception to extend an array?



```
Array Fixer

Input a word and type <ENTER>:  [        ]

these strings are stored in an ever growing array
```

```
           JPanel
┌─────────────────────────┐
│                         │
├─────────────────────────┤
│                         │
├─────────────────────────┤
│                         │
└─────────────────────────┘
              △
              │
        FixArrayBound
┌─────────────────────────┐
│ − list[] : String       │
│ − inField : JTextField   │
│ − prompt : JLabel        │
│ − count : int            │
├─────────────────────────┤
│ + FixArrayBound()        │
│ + paintComponent(in g : Graphics) │
│ − insertString(in s : String) │
│ + actionPerformed()      │
│ + main()                 │
└─────────────────────────┘
```
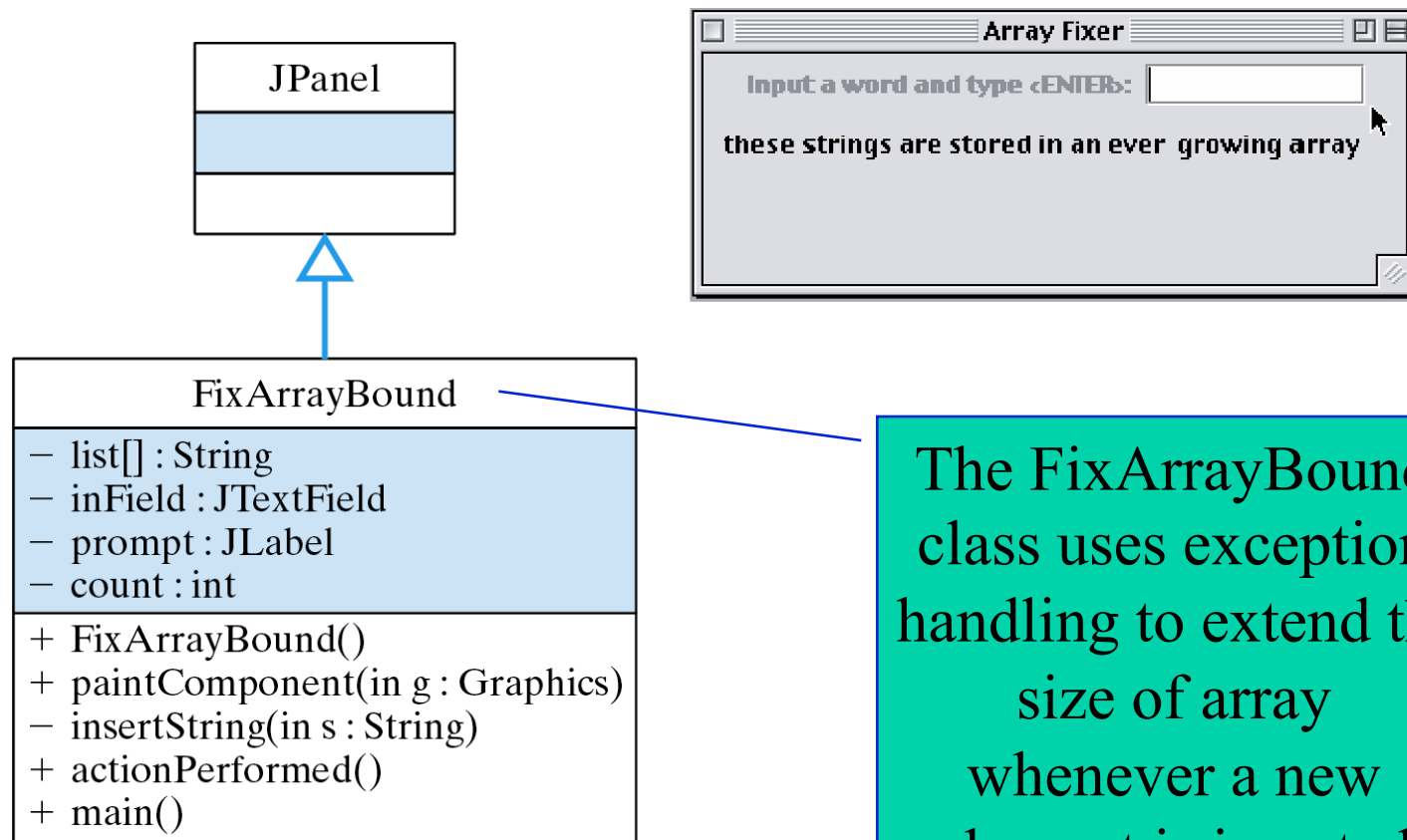
The FixArrayBound class uses exception handling to extend the size of array whenever a new element is inserted.

# When *Not* to Use an Exception

- **Effective Design:** Exceptions should *not* be used to handle routine conditions.
- If an array is routinely overflowed, use a Vector.

If array size is exceeded ...

```
private void insertString(String str) {
    try {
        list[count] = str;
    } catch (ArrayIndexOutOfBoundsException e) {
        String newList[] = new String[ list.length + 1 ]; // Create new array
        for (int k = 0; k < list.length ; k++)          // Copy old to new
            newList[k] = list[k];
            newList[count] = str;      // Insert item into new
            list = newList;                    // Make old point to new
    } finally {                                // Since the exception is now fixed
        count++;                               // Increase the count
    }
} // insertString()
```
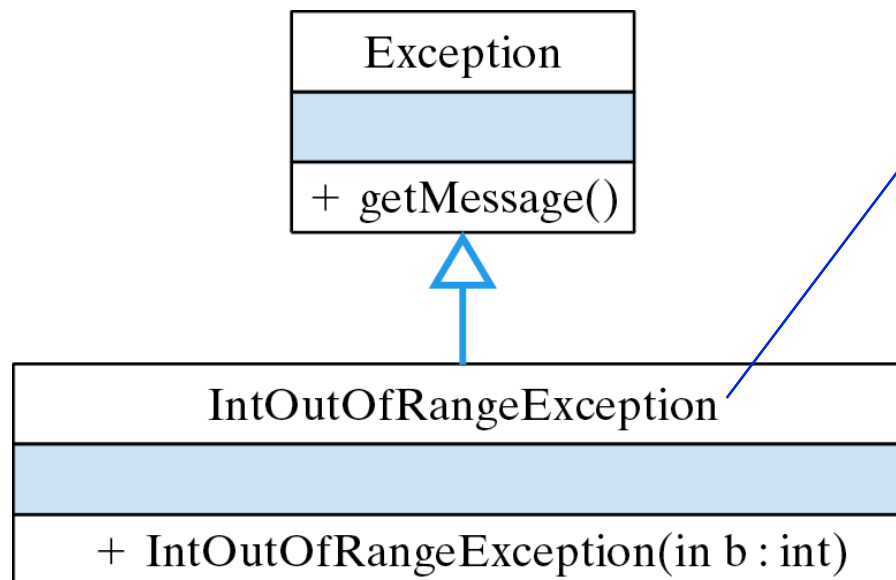
…extend its size.

- Source code: FixArrayBound.java

# Programmer-Defined Exceptions

- Programmer-defined exceptions are defined by extending the Exception class.

| Exception |
|---|
|  |
| + getMessage() |

| IntOutOfRangeException |
|---|
|  |
| + IntOutOfRangeException(in b : int) |

Thrown when an integer exceeds a certain bound.

# Creating Your Own Exceptions

- An exception for validating that an integer is less than or equal to a certain maximum value:
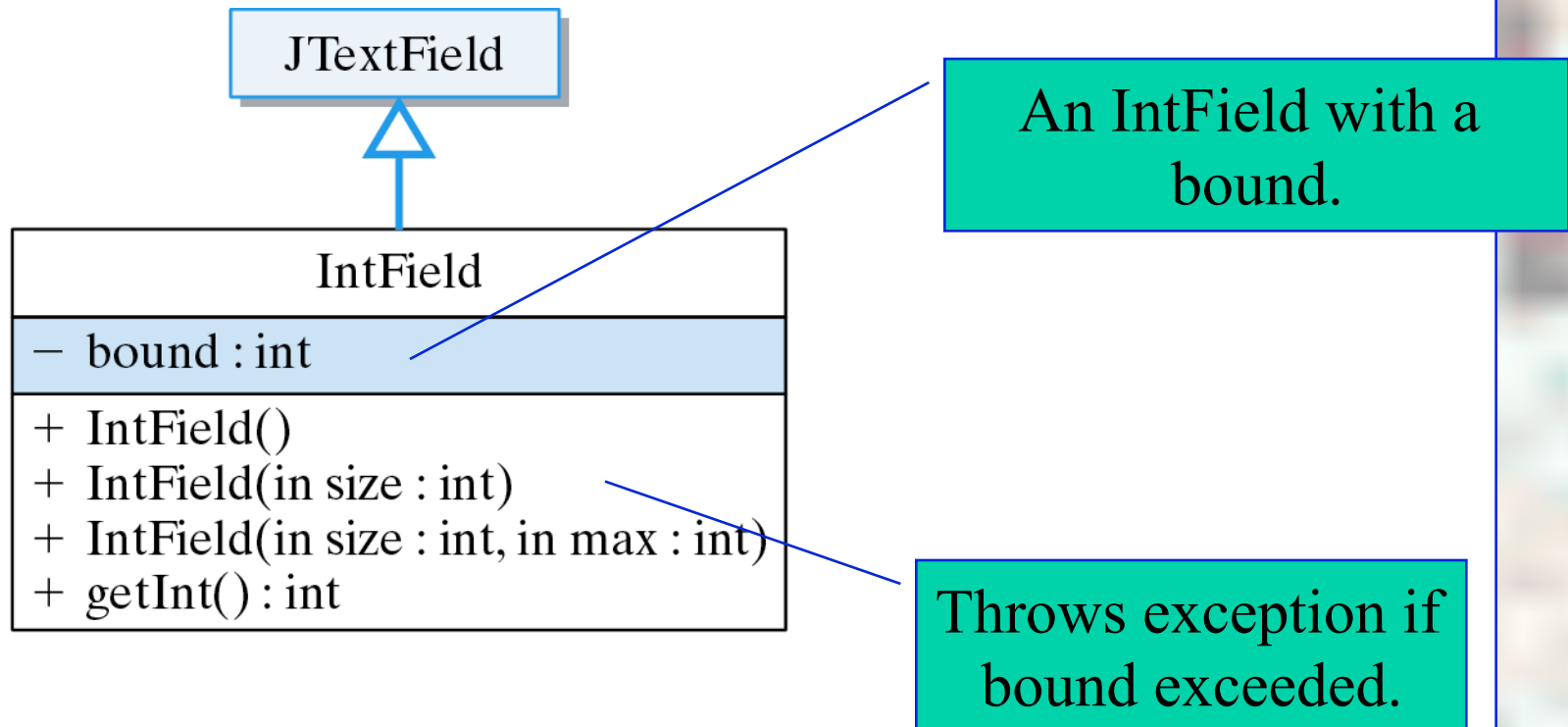
```java
/**
 *   IntOutOfRangeException reports an exception when an
 *      integer exceeds its bound.
 */
public class IntOutOfRangeException extends Exception {

    public IntOutOfRangeException (int Bound) {
        super("The input value exceeds the bound " + Bound);
    }
}
```

This error message will be printed when this exception is thrown.

# Example: Bounded Input

- Modified **IntField** that only accepts integers that are less than a certain bound.



JTextField

IntField

− bound : int

+ IntField()
+ IntField(in size : int)
+ IntField(in size : int, in max : int)
+ getInt() : int

An IntField with a bound.

Throws exception if bound exceeded.

# Implementation: IntField

- Bound is set in the IntField constructor.

```java
public class IntField extends JTextField {
    private int bound = Integer.MAX_VALUE;
    public IntField(int size, int max) {
        super(size);
        bound = max;
    }
    public int getInt() throws NumberFormatException,
                              IntOutOfRangeException {
        int num = Integer.parseInt(getText());
        if (num > bound)
            throw new IntOutOfRangeException(bound);
        return num;
    } // getInt()
      // The rest of the class is unchanged
} // IntField
```

New constructor lets us set the bound.

Throw exception if bound exceeded.

# Using Your Own Exception

- The **IntFieldTester** class tries to input an integer within a certain range:

```java
public class IntFieldTester extends JPanel implements ActionListener
{
    // Code deleted here
    public void actionPerformed(ActionEvent evt) {
        try {
            userInt = intField.getInt();
            message = "You input " + userInt + " Thank you.";
        } catch (NumberFormatException e) {
            JOptionPane.showMessageDialog(this,
                "The input must be an integer. Please reenter.");
        } catch (IntOutOfRangeException e) {
            JOptionPane.showMessageDialog(this, e.getMessage());
        } finally {
            repaint();
        }
    } // actionPerformed()
    // Code deleted here
} // IntFieldTester
```
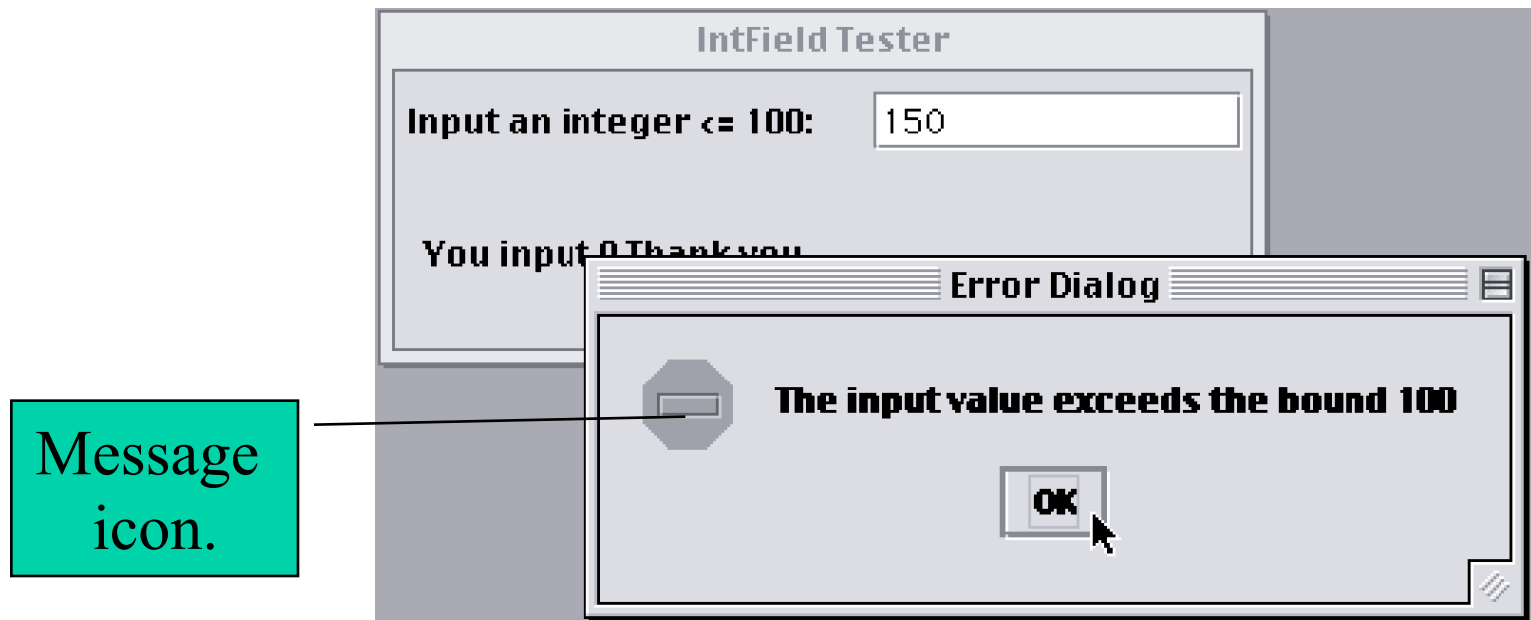
Get user's input.

Handle exceptions.

An error dialog window.

- Applet Version Demo: IntField Tester

# From the Java Library: JOptionPane

- A JOptionPane is a dialog box.
- A *file dialog* is used when you open or save a file. A *warning dialog* or *error dialog* communicates an error or warning message.



Message icon.

# JOptionPane

- ## JOptionPane.showMessageDialog():

```
catch (NumberFormatException e) {
    JOptionPane.showMessageDialog(this,
        "The input must be an integer. Please reenter.");
}
```

TopLevelWindow    Creates    →    DialogWindow
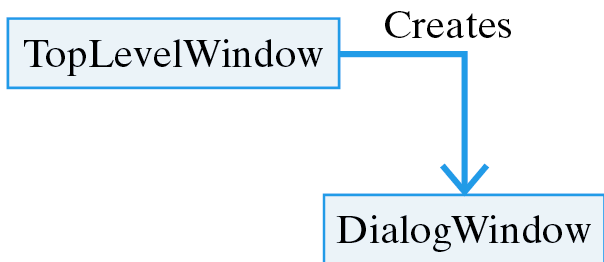
- ## Standard *message icons*:

```
JOptionPane.PLAIN_MESSAGE
JOptionPane.INFORMATIONAL_MESSAGE // Default
JOptionPane.WARNING_MESSAGE
JOptionPane.QUESTION_MESSAGE
JOptionPane.ERROR_MESSAGE
```

- ## Types of dialogs:

```
Message Dialog       Presents a simple error or informational message
Confirm Dialog       Prompts the user to confirm a particular action
Option Dialog        Lets the user choose from a couple of options
Input Dialog         Prompts and inputs a string
```

# Effective Design

- **Unfixable Error.** If possible, it's better to terminate the program abnormally than to allow the error to propagate.

- **Normal versus Exceptional Code.** The exception handler --- the catch block --- is distinct from the (normal) code that throws the exception --- the try block.

- **Using an Exception.** If your exception handler is not significantly different from Java's, let Java handle it.

# Effective Design

- **Handling Exceptions.**
  - Report the exception and terminate the program;
  - Fix the exceptional condition and resume normal execution.
  - Report the exception to a log and resume execution.
- **Program Development.** Exceptions help identify design flaws during program development.
- **Report and Resume.** Failsafe programs should report the exception and resume.

# Effective Design

- **Defensive Design.** Anticipate potential problems, especially potential input problems.

- **Fixing an Exception.** Handle fixable exceptions locally. This is both clearer and more efficient.

- **Library Exception Handling**. Many library classes leave exception handling to the application.

- **Truly Exceptional Conditions.** Use exceptions to handle truly exceptional conditions, not for expected conditions.

# Technical Terms

- catch block
- catch an exception
- checked exception
- dialog box
- dynamic scope
- error dialog
- exception
- exception handler

- finally block
- method call stack
- method stack trace
- modal dialog
- static scope
- throw an exception
- try block
- unchecked exception

# Summary Of Important Points

- In Java, when an error occurs, you throw an Exception which is caught by *exception handler* code . A *throw statement* --- throw new Exception () --- is used to throw an exception.

- A *try block* is contains one or more statements that may throw an exception. Embedding a statement in a try block indicates your awareness that it might throw an exception and your intention to handle the exception.

# Summary Of Important Points (cont)

- *Checked exceptions* must be caught or declared by the method in which they occur.
- *Unchecked exceptions* (subclasses of RuntimeException) are handled by Java if they are not caught in the program.
- A *catch block* contains statements that handle the exception that matches its parameter.
- A catch block can only follow a try block.
- There may be more than one catch block for each try block.

# Summary Of Important Points (cont)

- The try/catch syntax separates the normal parts of an algorithm from special exceptional handling code.

- A *method stack trace* is a trace of a program's method calls -- Exception.printStackTrace().

- *Static scoping:* how the program is written. Depends on declarations and definitions.

- *Dynamic scoping:* how the program is executed. Depends on method calls.

# Summary Of Important Points (cont)

- Finding a Catch Block: Search upward through the static scope, and backward through the dynamic scope.

- The Java Virtual Machine handles unchecked exceptions not caught by the program.

- Many Java library methods throw exceptions when an error occurs.

- Example: Java's integer division operator will throw an ArithmeticException if an attempt is made to divide by zero.

# Summary Of Important Points (cont)

- Four ways to handle an exception:
  - Let Java handle it.
  - Fix the problem and resume the program.
  - Report the problem and resume the program.
  - Print an error message and terminate.
- The (optional) finally block contains code that will be executed whether an exception is raised or not.
- Exceptions should be used for  exception truly exceptional conditions, not for normal program control.
- User-defined exceptions can extend the Exception class or one of its subclasses.