UCLA PIC 20A
Java Programming

- **Instructor:** Ivo Dinov,
  Asst. Prof. In Statistics, Neurology and Program in Computing
- **Teaching Assistant:** Yon Seo Kim, PIC

University of California, Los Angeles, Summer 2002

http://www.stat.ucla.edu/~dinov/

Chapter 7 – Runtime Errors
Exception Handling

- What Is an Exception?
- Catching and Handling Exceptions
- The try-catch-finally Block
- Exceptions Thrown by a Method
- Creating Your Own Exception Classes
- Why Use Exceptions?
- Examples

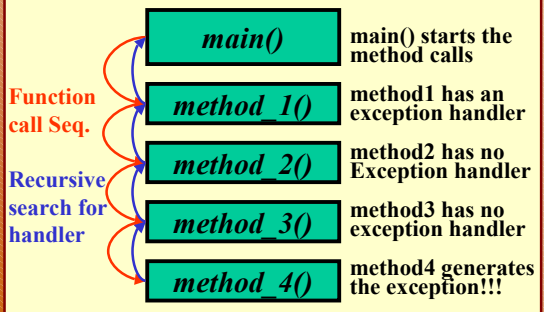
What are Exceptions?

- An *exception* is an exceptional event that disrupts the normal flow of instructions during the execution of a program.
- When a runtime 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 **state of the program**, **when** the error occurred. Creating an exception object and handing it to the JVM is called *throwing an exception*.

What are Exceptions?

- A method throws an exception ➔ Runtime system attempts to find something to handle it.
- The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred.
- The list of methods is known as the ***call stack.***

The Call Stack



Exception catch-or-specify requirement

- The Java VM requires that a method must either catch or specify all checked exceptions that can be thrown by that method.
- What are: "catch," "specify," "checked exceptions," and "exceptions that can be thrown by that method"?

Exception catch-or-specify requirement

- **Catch** – A method can catch an exception by providing an exception handler for that type of exception.
- **Specify** – A method specifies that it can throw exceptions by using the **throws** clause in the method declaration.
- **Checked exceptions** – There are two kinds of exceptions:
  - runtime exceptions and
  - non-runtime exceptions.

Exception catch-or-specify requirement

- **Runtime exceptions** occur within the Java runtime system: arithmetic exceptions (e.g., 1/0), pointer exceptions (e.g., null.member), and indexing exceptions (e.g., a = array[-1][Max+1];).
- **A method does not have to catch or specify runtime exceptions, although it may.**

Exception catch-or-specify requirement

- **Non-runtime exceptions** are exceptions that occur in code outside of the Java runtime system. For example, exceptions that occur during I/O are non-runtime exceptions.
- The compiler **requires** that non-runtime exceptions are caught or specified; hence *checked exceptions*.

Exception catch-or-specify requirement

- **Exceptions that can be thrown by a method** include:
  - Any exception thrown directly by the method with the **throw** statement
  - Any exception thrown indirectly by calling another method that throws an exception

Exception catching and handling

- **Exception handling mechanism** — the **try**, **catch**, and **finally** blocks
- The following example defines and implements a class named **ListOfNumbers**. Which creates a Vector that contains ten Integer elements numbered 0-9.
- The **ListOfNumbers** class also defines a method named **writeList** that writes the list of numbers into a text file called **OutFile**.txt.

Exception catching and handling – Ex.

```
public class ListOfNumbers {

    private Vector vec;
    private static final int SIZE =10;

    public ListOfNumbers ()
    {     vec  =  new Vector(SIZE);
          for (int i =0;  i <SIZE;  i++)
             vec.addElement(new Integer(i));
    }
```

Exception catching and handling – Ex.

```
public class ListOfNumbers {
        ....
        public void writeList()
        {       PrintWriter out =new PrintWriter(new
                        FileWriter("OutFile.txt"));
                for (int i =0; i  < SIZE;  i++)
                        out.println("Value at:“ + i + "=“ +
                                vec.elementAt(i) );
                out.close();
        }
}
```


Exception catching and handling – Ex.

- The call to **PrintWriter** constructor initializes an output stream on a file. If the file cannot be opened, the constructor throws an **IOException** .
- The Vector class's **elementAt** method, which throws an **ArrayIndexOutOfBoundsException** if the value of its argument is too small (<0) or too large (>Max).
- Trying to compile **ListOfNumbers** class generates **1** error message about the exception thrown by the **FileWriter** constructor –

Exception catching and handling – Ex.

- However, it does *not* display an error message about the exception thrown by **elementAt** , runtime exception (**ArrayIndexOutOfBoundsException**). Whereas the exception thrown by the constructor, (**IOException**), is a checked exception.

Exception handling – try block

- Exception handling is:
  - Enclose the statements that might throw an exception within a try block. In general:

    try {
    statements
    }

    statements may throw an exception, itself.
  - There is many ways to do this. Putting each statement that might throw an exception within its own try block and provide separate exception handlers for each. Or, putting all the writeList statements within a single try block.

Exception handling – try block

```
PrintWriter out =null;
   try {
      System.out.println("Entered try statement");
      out =new PrintWriter(new
              FileWriter("OutFile.txt"));
      for (int i =0; i  < size;  i++)
      out.println("Value at:“ +i + "=“ + vec.elementAt(i));
   }
```

If an exception occurs within the try block, that exception is handled by an exception handler associated with it. To associate an exception handler with a try block, put a catch statement after it.

Exception handling – catch block

You associate exception handlers with a try block by providing one or more catch blocks directly after the try .

```
 try {
            ...
 }  catch (ExceptionType name )   {
            ...
 }  catch (ExceptionType name )   {
            ...
 }          ...
```

Each catch block is an exception handler and handles the type of exception indicated by its argument.

Exception handling – catch block

Two exception handlers for **writeList** method

```
try {
    ...
} catch (ArrayIndexOutOfBoundsException e) {
  System.err.println("Caught
     ArrayIndexOutOfBoundsException:"
          + e.getMessage() );
} catch (IOException e) {
  System.err.println("Caught
          IOException:" + e.getMessage() );
}
```


Exception handling – catch block

- The handlers shown print an error message. Although simple, this might be the behavior you want.
- The exception gets caught, the user is notified, and the program continues to execute.
- However, exception handlers can do more. They can do error recovery, prompt the user to make a decision, or decide to exit the program.

Format for HW Project 6

- Open-ended:
  - You pick a (practical but simple) project that you write the Java package to solves the problem
  - E.g.'s,
    - Get a list of *.gif files from the user and animate them as a movie 500 milliseconds apart
    - Graphics Package – draw an object of interest (circle, rectangle, line, cube, star, etc.) with pre-defined parameters (center, size, line-width, etc.) with a desired color
    - Bouncing-ball application
    - Scientific Calculator
    - EmployeeRecordDatabase with file I/O
- Completely structured:
  - I select one problem that everyone works on!

Either way we'll want to use OOP design, exception handling, GUI widgets, packaging, etc.

Exception handling – finally block

- Last step in setting up an exception handler is to clean up before allowing control to be passed to a different part of the program. This is done by a **finally** block.
- The finally block is optional and provides a mechanism to clean up regardless of what happens within the try block.
- Ex., in exception occurring in the call to **PrintWriter**. The program should close that stream before exiting the **writeList** method.

Exception handling – finally block

- This poses a somewhat complicated problem because **writeList**'s try block can exit in one of three ways.
  - The new **FileWriter** statement fails and throws an **IOException** .
  - The **vec.elementAt(i)** statement fails and throws an **ArrayIndexOutOfBoundsException**.
  - Everything succeeds and the try block exits normally.

Exception handling – finally block

- The runtime system always executes the statements within the finally block regardless of what happens within the try block.

```
finally {
    if (out !=null){
        System.out.println("Closing
            PrintWriter");
        out.close();
    } else
        System.out.println("PrintWriter
            not open");
}
```

Exception handling

- The try block in this method has three exit possibilities.
  - The new **FileWriter** statement fails and throws an **IOException** .
  - The **vec.elementAt(i)** statement fails and throws an **ArrayIndexOutOfBoundsException**.
  - Everything succeeds and the try statement exits normally.
- Let's look at what happens in the **writeList** method during each of these exit possibilities.

Specifying Exceptions thrown by methods

```
public void writeList() {
        PrintWriter out =new PrintWriter(new
                FileWriter("OutFile.txt"));
        for (int i =0; i <size; i++)
                out.println("Value at:"+i
                        +"="+vec.elementAt(i));
        out.close();
}
```

To specify that writeList can throw two exceptions, you add a throws clause to the method declaration for the **writeList** method.

Specifying Exceptions thrown by methods

- The throws clause comprises the throws keyword followed by a comma-separated list of all the exceptions thrown by that method.
- The clause goes after the method name and argument list and before the brace that defines the scope of the method.

```
public void writeList() throws IOException,
    ArrayIndexOutOfBoundsException {
        …
}
```

Remember that ArrayIndexOutOfBoundsException is a runtime exception, so you don't have to specify it in the throws clause.

Throwing Exceptions

- Sometimes, it's appropriate for your code to catch exceptions that can occur within it. In other cases, however, it's better to let a method farther up the call stack handle the exception.
- For example, if you were providing the **ListOfNumbers** class as part of a package of classes, you probably couldn't anticipate the needs of all the users of your package.
- In this case, it's better to *not* catch the exception and to allow a method farther up the call stack to handle it.

Throwing Exceptions

```
public void writeList()   throws IOException,
        ArrayIndexOutOfBoundsException {
    PrintWriter out =new PrintWriter(new
        FileWriter("OutFile.txt"));
    for (int i = 0; i  < size; i++)
        out.println("Value at:"+i
                +"="+vec.elementAt(i));
    out.close();
}
```

writeList() thows two exceptions (**IOE**, **AIOBE**).

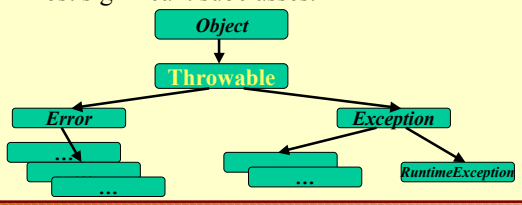
The throw clause

- The throw statement requires a single argument: a *throwable* object, instance of any subclass of the Throwable class.
- **Ex.** The method removes the top element from the stack and returns the object:

```
public Object  pop()  throws  EmptyStackException  {
        Object obj;
        if (size == 0)
                throw new EmptyStackException();
        obj =objectAt(size -1); setObjectAt(size -1,null);
        size--; return obj;
}
```

The throw clause

- The objects that inherit from the **Throwable** class include direct descendants and indirect descendants (objects that inherit from children or grandchildren of the Throwable class).
- Most significant subclasses.



Exceptions vs. Errors

- When a dynamic linking failure or other "hard" failure in the Java VM occurs, the Java VM throws an **Error** . Typical programs should not catch **Errors**. Also, typical programs never throw **Errors**.
- Most programs throw and catch objects that derive from the **Exception** class. An Exception indicates that a non serious system problem occurred.
- The **Exception** class has many descendants defined in the Java platform. E.g., **IllegalAccessException** signals that a particular method could not be found, and **NegativeArraySizeException** indicates an attempted to create an array with a negative size.

Extending the Exception class

- You may use other's Exception classes, but consider writing your own exception classes if:
  - Do you need an exception type that isn't represented by those in the Java platform?
  - Would it help your users if they could differentiate your exceptions from those thrown by classes written by others?
  - Does your code throw many related exceptions?
  - Will your users have access to those exceptions if you're using others Exceptions?
  - Should your package be independent and self-contained?

Why use Exceptions?

- 1. To separate the details of what to do when something out of the ordinary happens. E.g.,

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

  - Are there any Potential errors?
    - What happens if the file can't be opened?
    - What happens if the length of the file can't be determined?
    - What happens if enough memory can't be allocated?
    - What happens if the read fails?
    - What happens if the file can't be closed?

Why use Exceptions?

1. To separate the details of what to do when something out of the ordinary happens. E.g.,

```
readFile { try {   open the file;
                   determine its size;
                   allocate that much memory;
                   read the file into memory;
                   close the file;
         } catch (fileOpenFailed) doSomething1;
         catch (sizeDeterminationFailed)
                doSomething2;
         catch (memoryAllocFailed) doSomething3;
         catch (readFailed) doSomething4;
         catch (fileCloseFailed) doSomething5;
}
```


Why use Exceptions?

2. To allow error propagation up the call-stack. E.g.,

```
method1 {
  try {   call method2;
  } catch (Exception e)  doErrorProcessing;
}
method2 throws Exception {
   call method3;
}
method3 throws Exception {
   call readFile;
}
```

Why use Exceptions?

3. To allow **grouping** / **differentiating** error types. E.g.,

This handler will catch all I/O exceptions, including FileNotFoundException, EOFException, and so on.

```
catch (IOException e)  {
   e.printStackTrace(); //output goes to System.err
   e.printStackTrace(System.out);//send trace to stdout
}
```

This handler handles any Exception

```
catch (Exception e){ // a (too) general exception handler
...
}
```


Why use Exceptions?

3. To allow **grouping** / **differentiating** error types. E.g.,

This handler will catch all I/O exceptions, including FileNotFoundException, EOFException, and so on.

```
catch (IOException e)  {
   e.printStackTrace(); //output goes to System.err
   e.printStackTrace(System.out);//send trace to stdout
}
```

This handler handles any Exception

```
catch (Exception e){ // a (too) general exception handler
...
}
```

Slide 38 PIC 20A, UCLA, Ivo Dinov

An Exception Handling Example: Divide by Zero

- Example program
  - User enters two integers to be divided
  - We want to catch division by zero errors
  - Exceptions
    - Objects derived from class `Exception`
  - Look in `Exception` classes in `java.lang`
    - Nothing appropriate for divide by zero
    - Closest is `ArithmeticException`
    - Extend and create our own exception class

An Exception Handling Example: Divide by Zero

```
5  public class DivideByZeroException
6                extends ArithmeticException {
7     public DivideByZeroException()
12    public DivideByZeroException( String message )
```

- Two constructors for most exception classes
  - One with no arguments (default), with default message
  - One that receives exception message
  - Call to superclass constructor
- Code that may throw exception in `try` block
  - Covered in more detail in following sections
- Error handling code in `catch` block
- If no exception thrown, `catch` blocks skipped


```
1  // DivideByZeroException.java
2  // Definition of class DivideByZeroException.
3  // Used to throw an exception when a
4  // divide-by-zero is attempted.
5  public class DivideByZeroException
6                extends ArithmeticException {
7     public DivideByZeroException()
8     {
9        super( "Attempted to divide by zero"
10    }
11
12    public DivideByZeroException( String message )
13    {
14       super( mess
15    }
16 }
```

Define our own exception class (exceptions are thrown objects).

Default constructor (default message) and customizable message constructor.

- 1. Class `DivideByZero Exception` (`extends Arithmetic Exception`)
- 1.2 Constructors
- 1.3 `super`


```
18 // DivideByZeroTest.java
19 // A simple exception handling example.
20 // Checking for a divide-by-zero-error.
21 import java.text.DecimalFormat;
22 import javax.swing.*;
23 import java.awt.*;
24 import java.awt.event.*;
25
26 public class DivideByZeroTest extends JFrame
27                                implements ActionListener {
28    private JTextField input1, input2, output;
29    private int number1, number2;
30    private double result;
31
32    // Initialization
33    public DivideByZeroTest()
34    {
35       super( "Demonstrating Exceptions" );
36
37       Container c = getContentPane();
38       c.setLayout( new GridLayout( 3, 2 ) );
39
40       c.add( new JLabel( "Enter numerator ",
41                          SwingConstants.RIGHT ) );
42       input1 = new JTextField( 10 );
43       c.add( input1 );
44
45       c.add(
46          new JLabel( "Enter denominator and press Enter ",
47                      SwingConstants.RIGHT ) );
```

```
48      input2 = new JTextField( 10 );
49      c.add( input2 );
50      input2.addActionListener( this );
51
52
53
54      c.add( new JLabel( "RESULT ",
55      output = new JTextField();
56      c.add( output );
57
58      setSize( 425, 100 );
59      show();
60   }
61
62   // Process GUI events
63   public void actionPerformed( Actio
64   {
65      DecimalFormat precision3 = new
66
67      output.setText( "" ); // empty the output
68
69      try {
70         number1 = Integer.parseInt( input1.getText()
71         number2 = Integer.parseInt( input2.getText()
72
73         result = quotient( number1, number2 );
74         output.setText( precision3.format( result )
75      }
```

Notice enclosing **try** block. If an exception is thrown in the block (even from a method call), the entire block is terminated.

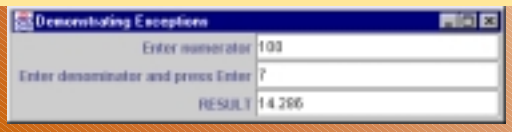

```
76      catch ( NumberFormatException nfe ) {
77         JOptionPane.showMessageDialog( this,
78            "You must enter two integers",
79            "Invalid Number Format",
80            JOptionPane.ERROR_MESSAGE );
81      }
82      catch ( DivideByZeroException dbze ) {
83         JOptionPane.showMessageDialog( this,
84            "Attempted to Divide by Zero",
85            JOptionPane.ERROR_MESSAGE );
86      }
87   }
88
89   // Definition of method quotient. Used to
90   // throwing an exception when a divide-by-zero
91   // is encountered.
92   public double quotient( int numerator, int
93      throws DivideByZeroException
94   {
95      if ( denominator == 0 )
96         throw new DivideByZeroException();
97
98      return ( double ) numerator / denominator;
99   }
100
101      public static void main( String argsf] )
102      {
103         DivideByZeroTest app = new
104
105
```

catch blocks have error handling code. Control resumes after the **catch** blocks.

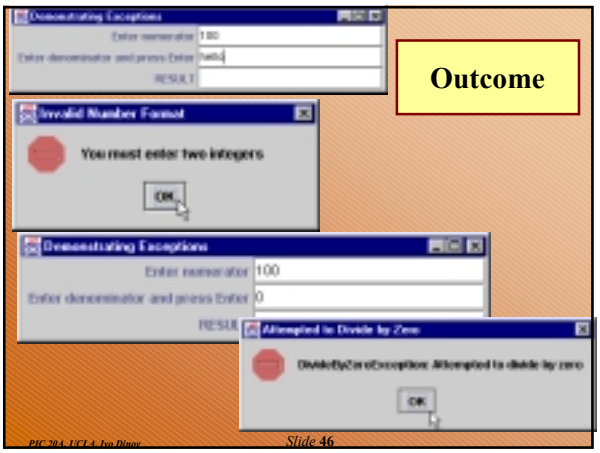
The first block makes sure the inputs are of the correct type.

Method **quotient** throws an **DivideByZeroException** exception (object) if **denominator == 0**.


```
106         app.addWindowListener(
107            new WindowAdapter() {
108               public void windowClosing( WindowEvent e )
109               {
110                  e.getWindow().dispose();
111                  System.exit( 0 );
112               }
113            }
114         );
115      }
116   }
```



Outcome



Try Blocks

- Exceptions that occurs in a **try** block
  - Usually caught by handler specified by following **catch** block

```
try{
   code that may throw exceptions
}
catch ( ExceptionType ref ) {
   exception handling code
}
```

    - Can have any number of **catch** blocks
  - If no exceptions thrown, **catch** blocks skipped

Throwing an Exception

- **throw**
  - Indicates exception has occurred (throwing an exception)
  - Operand
    - Object of any class derived from **Throwable**

```
95      if ( denominator == 0 )
96         throw new DivideByZeroException();
```

  - Derived from **Throwable:**
    - **Exception** - most programmers deal with
    - **Error** - serious, should not be caught
- When exception thrown
  - Control exits current **try** block
  - Proceeds to **catch** handler (if exists)

Throwing an Exception

- Exceptions
  - Can still throw exceptions without explicit **`throw`** statement
  - **`ArrayIndexOutOfBoundsException`**
  - Terminates block that threw exception
    - Not required to terminate program

Catching an Exception

- **`catch`** blocks
  - Contain exception handlers
  - Format:

```
catch( ExceptionType ref ) {
   error handling code
}
```

```
82      catch ( DivideByZeroException dbze ) {
83         JOptionPane.showMessageDialog( this,
84            "Attempted to Divide by Zero",
85            JOptionPane.ERROR_MESSAGE );
86      }
```

  - To catch all exceptions, catch an exception object:
    `catch( Exception e )`

Catching an Exception

- Catching exceptions
  - First handler to catch exception does
    - All other handlers skipped
  - If exception not caught
    - Searches enclosing **`try`** blocks for appropriate handler

```
try{
   try{
      throw Exception2
   }
   catch ( Exception1 ){...}
}
catch( Exception2 ){...}
```

  - If still not caught, non-GUI based applications terminate

Catching an Exception

- Information
  - Information can be passed in the thrown object
  - Data in instance variables
- If a **`catch`** block throws an exception
  - Exception must be processed in enclosing **`try`** block
- Usage of exception handlers
  - Rethrow exception (next section)
    - Convert exception to different type
  - Perform recovery and resume execution
  - Look at situation, fix error, and call method that generated exception
  - Return a status variable to environment

Rethrowing an Exception

- Rethrowing exceptions
  - Use if handler cannot process exception
  - Rethrow exception with the statement:
    `throw e;`
    - Detected by next enclosing **`try`** block
  - Handler can always rethrow exception, even if it performed some processing

Throws Clause

- Throws clause
  - Lists exceptions that can be thrown by a method

```
92    public double quotient( int numerator, int denominator )
93       throws DivideByZeroException
```

```
int g( float h ) throws a, b, c
{
   // method body
}
```

  - Method can throw listed exceptions or derived types

Throws Clause

- Run-time exceptions
  - Derive from **`RunTimeException`**
  - Some exceptions can occur at any point
    - **`ArrayIndexOutOfBoundsException`**
    - **`NullPointerException`**
      - Create object reference without attaching object to reference
    - **`ClassCastException`**
      - Invalid casts
  - Most avoidable by writing proper code

Throws Clause

- Checked exceptions
  - Must be listed in **`throws`** clause of method
  - All non-**`RuntimeException`**s
- Unchecked exceptions
  - Can be thrown from almost any method
    - Tedious to write **`throws`** clause every time
    - No **`throws`** clause needed
  - **`Error`**s and **`RunTimeException`**s

Throws Clause

- Catch-or-declare requirement
  - If method calls another method that explicitly throws checked exceptions
    - Exceptions must be in original method's **`throws`** clause
  - Otherwise, original method must **`catch`** exception
  - Method must either **`catch`** exception or declare it in the **`throw`** clause

Constructors, Finalizers and Exception Handling

- What to do with an error in constructor?
  - Constructor cannot return value
  - How do we inform program of error?
  - Possible solutions
    - Hope someone tests defective object
    - Set some variable outside constructor
  - Thrown exception informs program of a failed constructor
- Exceptions thrown in constructors
  - Object marked for garbage collection
    - **`finalize`**
  - No particular order

Exceptions and Inheritance

- Inheritance
  - Exception classes can have a common superclass
  - **`catch ( Superclass ref )`**
    - Catches subclasses
    - "Is a" relationship
  - Polymorphic processing
  - Easier to catch superclass than catching every subclass

`finally` Block

- Resource leaks
  - Programs obtain and do not return resources
  - Automatic garbage collection avoids most memory leaks
    - Other leaks can still occur
- **`finally`** block
  - Placed after last **`catch`** block
  - Can be used to returns resources allocated in **`try`** block
  - Always executed, irregardless whether exceptions thrown or caught
  - If exception thrown in **`finally`** block, processed by enclosing **`try`** block
    - If there was an original exception, it is lost

```
// UsingExceptions.java
// Demonstration of stack unwinding.
public class UsingExceptions {
   public static void main( String args[] )
   {
      try {
         throwException();
      }
      catch ( Exception e ) {
         System.err.println( "Exception ha
      }
   }

   public static void throwException() throws Exception
   {
      // Throw an exception and catch it in main.
      try {
         System.out.println( "Method throwException" );
         throw new Exception();       // generate
      }
      catch( RuntimeException e ) {  // nothing caught
         System.err.println( "Exception handled in " +
                             "method throwException" );
      }
      finally {
         System.err.println( "Finally is always
      }
   }
}
```

Call method **throwException** (enclosed in a **try** block).

Throw an **Exception**. The **catch** block cannot handle it, but the **finally** block executes irregardless.

Using `printStackTrace` and `getMessage`

- Class **Throwable**
  - Superclass of all exceptions
  - Method **printStackTrace**
    - Prints method call stack for caught **Exception** object
      - Most recent method on top of stack
    - Helpful for testing/debugging
  - Constructors
    - **Exception()**
    - **Exception(String informationString)**
  - **informationString** may be accessed with method **getMessage**


```
// UsingExceptions.java
// Demonstrating the getMessage and printStackTrace
// methods inherited into all exception classes.
public class UsingExceptions {
   public static void main( String args[] )
   {
      try {
         method1();
      }
      catch ( Exception e ) {
         System.err.println( e.getMessage() + "\n" );

         e.printStackTrace();
      }
   }

   public static void method1() throws Exception
   {
      method2();
   }

   public static void method2() t
   {
      method3();
   }

   public static void method3() t
   {
      throw new Exception( "Exception thrown in
   }
}
```

Call **method1**, which calls **method2**, which calls **method3**, which throws an exception.

getMessage prints the **String** the **Exception** was initialized with.

printStackTrace prints the methods in this order:

method3
method2
method1
main

(order they were called when exception occurred)


```
Exception thrown in method3
java.lang.Exception: Exception thrown in method3
   at UsingExceptions.method3(UsingExceptions.java:28)
   at UsingExceptions.method2(UsingExceptions.java:23)
   at UsingExceptions.method1(UsingExceptions.java:18)
   at UsingExceptions.main(UsingExceptions.java:8)
```

- Program Output