COP 3330: Object-Oriented Programming Summer 2010

Exception Handling In Java

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2010

School of Electrical Engineering and Computer Science
University of Central Florida

Exception Handling In Java

- When a program encounters a runtime error, it terminates abnormally.
- What we would like as software developers, is for our programs to either continue execute or else terminate gracefully in the event of a run-time error.
- In Java an exception is an object created from an exception class.

Exception Handling In Java

- To demonstrate exception handling, consider the following code that reads two integers and computes their quotient. What happens when you enter 5 and 2? What happens when you enter 5 and 0?

PrintCalendarWithRea | PetDatabase.java | Quotient.java

```
// First example in Exception Handling notes
// COP 3330 - Summer 2011
// MJL  6/23/2011

import java.util.Scanner;

public class Quotient {
    public static void main (String args[]){
        Scanner input = new Scanner(System.in);
        //Prompt user to enter two integer values
        System.out.println ("Please enter two integer numbers...");
        System.out.print("Enter integer 1: ");
        int number1 = input.nextInt();
        System.out.print("Enter integer 2: ");
        int number2 = input.nextInt();
        System.out.println("\n" + number1 + " / " + number2 + " = "
                + (number1/number2));
    }//end main method
}//end class Quotient
```

Exception Handling In Java

```
Console
<terminated> Quotient [Java Application] C:\Program Files\Jav
Please enter two integer numbers...
Enter integer 1: 6
Enter integer 2: 3

6 / 3 = 2
```

Enter 6 and 3…no problem!

Run-time exception

```
Console
<terminated> Quotient [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 23, 2011 2:01:00 PM)
Please enter two integer numbers...
Enter integer 1: 6
Enter integer 2: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
	at Quotient.main(Quotient.java:17)
```

Enter 6 and 0…oops!

Exception Handling In Java

- A simple fix to the quotient code to fix this problem is shown below.

PrintCalendarWithRea | PetDatabase.java | Quotient.java | QuotientWithIfStmt.j

```java
// Second example in Exception Handling notes
// Possible fix to first example
// COP 3330 - Summer 2011
// MJL  6/23/2011

import java.util.Scanner;

public class QuotientWithIfStmt {
    public static void main (String args[]){
        Scanner input = new Scanner(System.in);
        //Prompt user to enter two integer values
        System.out.println ("Please enter two integer numbers...");
        System.out.print("Enter integer 1: ");
        int number1 = input.nextInt();
        System.out.print("Enter integer 2: ");
        int number2 = input.nextInt();
        if (number2 != 0)
            System.out.println("\n" + number1 + " / " + number2 + " = "
                + (number1/number2));
        else
            System.out.println("Divisor cannot be zero...Program terminates.");
    }//end main method
}//end class Quotient
```

Simple fix is to insert an if statement to make sure that the 2nd number is not zero.

Exception Handling In Java

Package Explorer
- Exam 1 - Summer 2011
- Exception Handling
  - src
    - (default package)
      - Quotient.java
      - QuotientWithIfStmt.ja
  - JRE System Library [JavaSE-1.(
- In Class Practice #1
- In Class Practice #2
- Interfaces

Quotient.java | QuotientWithIfStmt.java

```java
// Second example in Exception Handling notes
// Possible fix to first example
// COP 3330 - Summer 2011
// MJL  6/23/2011

import java.util.Scanner;

public class QuotientWithIfStmt {
    public static void main (String args[]){
        Scanner input = new Scanner(System.in);
        //Prompt user to enter two integer values
        System.out.println ("Please enter two integer numb
```

Console

<terminated> QuotientWithIfStmt [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 23, 2011 3:30:30 PM)

```
Please enter two integer numbers...
Enter integer 1: 6
Enter integer 2: 0
Divisor cannot be zero...Program terminates.
```

Exception Handling In Java

- We can rewrite the quick fix solution to our Quotient program using exception handling. This is shown on the next page.
- Notice that the exception handling version is somewhat larger than the simple fix version. This example is too small to actually achieve any serious benefit from exception handling and as such we would not typically employ exception handling in this sort of situation…this was an example only.

Quotient.java | QuotientWithIfStmt.java | *QuotientWithExceptionHandling.java

```java
// First example using Exception Handling
// Modification of the Quotient example
// COP 3330 - Summer 2011
// MJL  6/23/2011

import java.util.Scanner;

public class QuotientWithExceptionHandling {
    public static void main (String args[]){
        Scanner input = new Scanner(System.in);
        //Prompt user to enter two integer values
        System.out.println ("Please enter two integer numbers...");
        System.out.print("Enter integer 1: ");
        int number1 = input.nextInt();
        System.out.print("Enter integer 2: ");
        int number2 = input.nextInt();
        //start try block
        try {
            if (number2 == 0)
                throw new ArithmeticException("Divisor cannot be zero!");
            System.out.println("\n" + number1 + " / " + number2 + " = "
                + (number1/number2));
        }//end try block
        catch (Exception ex){
            System.out.println("Exception: An integer cannot" +
                    " be divided by zero.");
        }//end catch block
        System.out.println("Program execution continues at this point...");
    }//end main method
}//end class Quotient
```

try block

catch block

Exception Handling In Java

```
<terminated> QuotientWithExceptionHandling [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 23, 2011 3:39:03 PM)
Please enter two integer numbers...
Enter integer 1: 6
Enter integer 2: 0
Exception: An integer cannot be divided by zero.
Program execution continues at this point...
```

Output from the catch block that caught the thrown exception

Exception Handling In Java

- The `try` block contains the code that is executed in normal circumstances.
- The `catch` block contains the code that is executed when an exception occurs.
- In this case, the program throws an exception by executing

```
throw new ArithmeticException("Divisor cannot be zero");
```

- The value thrown, is called an exception.
- The execution of a throw statement is called throwing an exception.
- The exception is an object created from an exception class. In this case, the exception class is `java.lang.ArithmeticException`.

Exception Handling In Java

- When an exception is thrown, the normal execution flow is interrupted.
- As the name suggests, "throwing an exception" is to pass the exception from one place to another.
- The exception is caught by the `catch` block.
- The code in the catch block is executed to handle the exception. Afterward, the statement immediately after the catch block is executed (i.e., normal execution flow resumes).
- The `throw` statement is analogous to a method call, but instead of calling a method, it calls a `catch` block.

Exception Handling In Java

- In this sense, a `catch` block is like a method definition with a parameter that matches the type of the value being thrown.
- Unlike a method, after executing the `catch` block, the program control does not return back to the `throw` statement; instead, it executes the next statement after the `catch` block.
- The identifier `ex` in the catch block header:

```
catch (ArithmeticException ex)
```

acts very much like a parameter in a method. So this parameter is referred to as a `catch` block parameter.

Exception Handling In Java

- The type (e.g. `ArithmeticException`) preceding `ex` specifies what kind of exception the `catch` block can catch.
- Once the exception is caught, you can access the thrown value from this parameter in the body of a `catch` block.
- The following page shows a template for a generic `try-throw-catch` block.

Exception Handling In Java

```
try {
   code to try;
   throw an exception with a throw statement or from
   a method if necessary.
   more code to try;
}
catch (type ex) {
  code to process the exception;
}
```

generic `try-throw-catch` block

© Mark Llewellyn

Advantages of Exception Handling

- The biggest advantage of exception handling in Java is the ability it provides for a method to throw an exception back to its caller.
- Without this capability the method would be required to either handle the exception itself or to terminate the program.
- The following example illustrates this advantage.

Quotient.java | QuotientWithIfStmt.java | QuotientWithExceptionHa | QuotientWithMethod.java

```java
public class QuotientWithMethod {

    public static int quotient(int number1, int number2) {
        if (number2 == 0)
            throw new ArithmeticException("Divisor cannot be zero!");
        return number1/number2;
    }//end method quotient

    public static void main (String args[]){
        Scanner input = new Scanner(System.in);
        //Prompt user to enter two integer values
        System.out.println ("Please enter two integer numbers...");
        System.out.print("Enter integer 1: ");
        int number1 = input.nextInt();
        System.out.print("Enter integer 2: ");
        int number2 = input.nextInt();
        //start try block
        try {
            int result = quotient(number1, number2);
            System.out.println("\n" + number1 + " / " + number2 + " = "
                + (number1/number2));
        }//end try block
        catch (Exception ex){
            System.out.println("Exception Message #1: An integer cannot" +
                    " be divided by zero.");
            //the next output uses the parameter value passed from the exception meth
            System.out.println("Exception Message #2: " + ex);
        }//end catch block
        System.out.println("Program execution continues at this point...");
    }//end main method
}//end class QuotientWithMethod
```

Exception occurs here… thrown back to caller in main

Java - Exception Handling/src/QuotientWithMethod.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Console

<terminated> QuotientWithMethod [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 23, 2011 3:50:47 PM)

```
Please enter two integer numbers...
Enter integer 1: 6
Enter integer 2: 0
Exception Message #1: An integer cannot be divided by zero.
Exception Message #2: java.lang.ArithmeticException: Divisor cannot be zero!
Program execution continues at this point...
```

Exception Types

- The `catch` block parameter in the `QuotientWithMethod` example is of the `ArithmeticException` type.
- You can use the `Throwable` class or any subclass of `Throwable`. `ArithmeticException` is a subclass of `Throwable`.
- The `Throwable` class is contained in the `java.lang` package, and subclasses of `Throwable` are contained in various packages. Errors related to GUI components are included in the `java.awt` package; numeric exceptions are included in the `java.lang` package, because they are related to the `java.lang.Number` class.
- You can create your own exception classes by extending `Throwable` or a subclass of `Throwable`.
- The following page shows some of Java's predefined exception classes.

Exception Types

- Object
  - Throwable
    - Exception
      - ClassNotFoundException
      - IOException
      - AWTException
      - RuntimeException
        - ArithmeticException
        - NullPointerException
        - IndexOutOfBoundsException
        - IllegalArgumentException
        - Several more classes
      - Several more classes
    - Error
      - LinkageError
      - VirtualMachineError
      - AWTError
      - Several more classes

System Errors

System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Object
Throwable
Exception
ClassNotFoundException
IOException
AWTException
RuntimeException
Several more classes
ArithmeticException
NullPointerException
IndexOutOfBoundsException
IllegalArgumentException
Several more classes
Error
LinkageError
VirtualMachineError
AWTError
Several more classes

Exceptions

Exceptions describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.

ClassNotFoundException
IOException
AWTException
RuntimeException
Several more classes
ArithmeticException
NullPointerException
IndexOutOfBoundsException
IllegalArgumentException
Several more classes
Object
Throwable
Exception
Error
LinkageError
VirtualMachineError
AWTError
Several more classes

Runtime Exceptions

A RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

Object
Throwable
Exception
ClassNotFoundException
IOException
AWTException
RuntimeException
ArithmeticException
NullPointerException
IndexOutOfBoundsException
IllegalArgumentException
Several more classes
Several more classes
Error
LinkageError
VirtualMachineError
AWTError
Several more classes

Checked vs. Unchecked Exceptions

- `RuntimeException`, `Error`, and their subclasses are known as unchecked exceptions. All other exceptions are known as checked exceptions.
- Checked exceptions means that the compiler forces the programmer to check and deal with them.
- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example, a `NullPointerException` is thrown if you access an object through a reference variable before an object is assigned to it; an `IndexOutOfBoundsException` is thrown if you access an element in an array outside the bounds of the array.
- Unchecked exceptions can occur anywhere in a program. To avoid overuse of try-catch blocks, Java does not require the programmer write code to catch or declare unchecked exceptions.

Understanding Exception Handling

- Java's exception handling model is based on three operations: declaring an exception, throwing an exception, and catching an exception.

```
method1() {
  try {
      invoke method2;
  }

  catch (Exception ex) {
      process exception;
  }
}
```

catch exception

```
method2() throws Exception {

   if (an error occurs) {

      throw new Exception();

   }
}
```

declare exception

throw exception

Declaring Exceptions

- In Java, the statement currently being executed belongs to a method. The Java interpreter invokes the `main` method for a Java application (the Web browser invokes an applet's no-arg constructor and then the `init` method for a Java applet).
- Every method must state the types of checked exception it might throw. This is known as declaring exceptions. Exceptions must be explicitly declared in the method declaration so that the caller of the method is informed of the exception.
- To declare an exception in a method, use the `throws` keyword in the method declaration as shown:

```
public void myMethod() throws IOException
```

Declaring Exceptions

- The `throws` keyword indicates that `myMethod` might throw an `IOException`.
- If the method might throw multiple exceptions, add a list of the exceptions, separated by commas, after `throws`, such as:

```
public void myMethod()
 throws Exception1, Exception2,
    ...ExceptionN
```

- Note: if a method does not declare exceptions in the superclass, you cannot override it to declare exceptions in the subclass.

Throwing Exceptions

- A program that detects an error can create an instance of an appropriate exception type and throw it. This is known as throwing an exception.
- As an example, suppose the program detects that an argument passed to the method violates the method contract (e.g., the argument must be non-negative, but a negative argument is passed); the program can create an instance of `IllegalArgumentException` and throw it as follows:

```
throw new IllegalArgumentException("Wrong Argument");
```

- `IllegalArgumentException` is an exception class in the Java API. In general, each exception class in the Java API has at least two constructors, a no-arg constructor, and a constructor with a `String` argument that describes the exception. This argument is called the exception message, which can be obtained using `getMessage()`.

Catching Exceptions

- When an exception is thrown, it can be caught and handled in a `try-catch` block.

```
try {
    statements;  //statements that may throw exceptions
}
catch (Exception1 exVar1) {
   //handler for exception1;
}
catch (Exception2 exVar2) {
   //handler for exception2;
}
. . .
catch (ExceptionN exVarN) {
  //handler for exceptionN;
}
```

Catching Exceptions

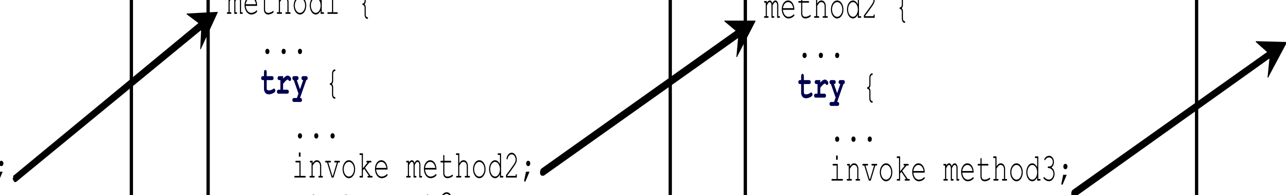
- If no exceptions arise during the execution of the `try` block, the `catch` blocks are skipped.
- If one of the statements inside the `try` block throws an exception, Java skips the remaining statements in the `try` block and starts the process of finding the code to handle the exception.
- The code that handles the exception is called the exception handler.
- The exception handler is found by propagating the exception backward through a chain of method calls, starting from the current method.
- Each `catch` block is examined in turn, from first to last, to see whether the type of the exception object is an instance of the exception class in the `catch` block.
- If so, the exception object is assigned to the variable declared, and the code in the catch block is executed.

Catching Exceptions

- If no handler is found, Java exits this method, passes the exception to the method that invoked that method, and continues the same process to find a handler.
- If no handler is found in the chain of methods being invoked, the program terminates and prints an error message on the console.
- This process is known as catching an exception.
- To better understand this scenario, consider the situation shown on the next two pages.
  - Suppose the main method invokes method1, method1 invokes method2, method2 invokes method3, and an exception occurs in method3.
  - If method3 cannot handle the exception, method3 is aborted and the control is returned to method2. If the exception type is Exception3, it is caught by the catch block for handling ex3 in method2. Statement5 is skipped, and statement6 is executed.
  - If the exception type is Exception2, method2 is aborted with control returning to method1, and the exception is caught by the catch block for handling ex2 in method1, statement3 is skipped and statement4 is executed.

Catching Exceptions

- Suppose the main method invokes method1, method1 invokes method2, method2 invokes method3, and an exception occurs in method3.
- If method3 cannot handle the exception, method3 is aborted and the control is returned to method2. If the exception type is Exception3, it is caught by the catch block for handling ex3 in method2. Statement5 is skipped, and statement6 is executed.
- If the exception type is Exception2, method2 is aborted with control returning to method1, and the exception is caught by the catch block for handling ex2 in method1, statement3 is skipped and statement4 is executed.
- If the exception type is Exception1, method1 is aborted with control returning to main, and the exception is caught by the catch block for handling exception ex1 in main, statement1 is skipped, and statement2 is executed.
- If the exception type is not Exception1, Exception2, or Exception3, the exception is not caught and the program terminates, statement1 and statement2 are not executed.



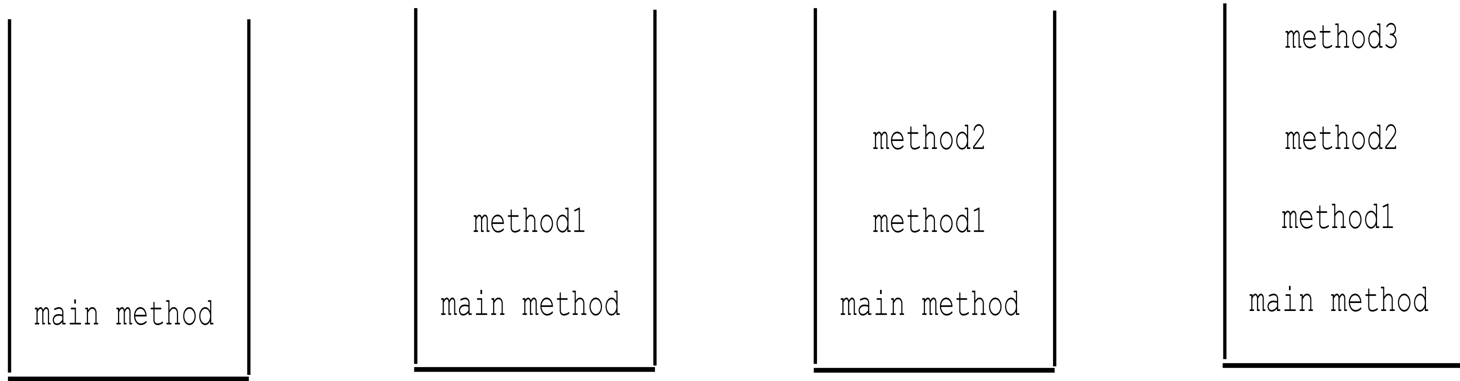
```
main method {
  ...
  try {
    ...
    invoke method1;
    statement1;
  }
  catch (Exception1 ex1) {
    Process ex1;
  }
  statement2;
}
```

```
method1 {
  ...
  try {
    ...
    invoke method2;
    statement3;
  }
  catch (Exception2 ex2) {
    Process ex2;
  }
  statement4;
}
```

```
method2 {
  ...
  try {
    ...
    invoke method3;
    statement5;
  }
  catch (Exception3 ex3) {
    Process ex3;
  }
  statement6;
}
```

An exception is thrown in method3

Call Stack



Catching Exceptions

- Various exceptions can be derived from a common superclass. If a `catch` block catches exception objects of a superclass, it can catch all the exception objects of the subclasses of that superclass.
- The order in which exceptions are specified in catch blocks is important. A compilation error will result if a catch block for a superclass type appears before a catch block for a subclass type.

Wrong order

```
try {
   . . .
}
catch (Exception ex) {
 . . .
}
catch (RunTimeException ex) {
  . . .
}
```

Correct order

```
try {
   . . .
}
catch (RunTimeException ex) {
 . . .
}
catch (Exception ex) {
  . . .
}
```

Catching Exceptions

- Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than `Error` or `RunTimeException`) you must invoke it in a try-catch block or declare to throw the exception in the calling method.
- For example, suppose method p1 invokes method p2 and p2 may throw a checked exception, then you must write the code as in one of the two options shown below:

```
void p1() {
   try {
      p2();
   }
   catch (IOException ex) {
      . . .
   }
}
```

catching the exception

```
void p1() throws IOException {
   . . .
   p2();
   . . .
}
```

throwing the exception

Getting Information From Exceptions

- An `Exception` object contains valuable information about the exception. You can use the following instance methods in the `java.lang.Throwable` class to get information regarding the exception. The example on the next page illustrates using an `Exception` object.

```
java.lang.Throwable
---------------------------------
+ getMessage(): String
+ toString(): String
+ printStackTrace(): void
+ getStackTrace():
      StackTraceElement[]
```

- `getMessage()`: Returns the message of this object
- `toString()`: Returns the concatenation of three strings: (1) the full name of the exception class; (2) ": " (a colon and a space; (3) the `getMessage()` method.
- `printStackTrace()`: Prints the Throwable object and its call stack trace information on the console.
- `getStackTrace()`: Returns an array of stack trace elements representing the stack trace pertaining to this throwable.

TestException.java

```java
// For Exception Handling Notes - COP 3330 - Summer 2011
// MJL  6-27/2011

public class TestException {

    private static int sum(int[] list) {
        int result = 0;
        for (int i = 0; i <= list.length; i++)
            result += list[i];
        return result;
    }//end method sum

    public static void main(String args[]) {
        try {
            System.out.println(sum (new int[] {1, 2, 3, 4, 5}));
        }//end try block
        catch (Exception ex) {
            ex.printStackTrace();
            System.out.println("\n" + ex.getMessage());
            System.out.println("\n" + ex.toString());
            System.out.println("\n Trace information obtained from getStackTrace:");
            StackTraceElement[]traceElements = ex.getStackTrace();
            for (int i = 0; i < traceElements.length; i++) {
                System.out.print("Method: " + traceElements[i].getMethodName());
                System.out.print("(" + traceElements[i].getClassName() + ":");
                System.out.println(traceElements[i].getLineNumber() + ")");
            }//end for stmt
        }//end catch block
    }//end main method
}//end class TestException
```

Console

<terminated> TestException [Java Application] C:\Program ... (Jun 27, 2011 1:10:53 PM)

printStackTrace()

```
java.lang.ArrayIndexOutOfBoundsException: 5
```

getMessage()

```
5
```

toString()

```
java.lang.ArrayIndexOutOfBoundsException: 5
```

using getStackTrace()

```
 Trace information obtained from getStackTrace:
Method: sum(TestException:9)
Method: main(TestException:15)
        at TestException.sum(TestException.java:9)
        at TestException.main(TestException.java:15)
```

Example: Declaring, Throwing and Catching Exceptions

- Going back to our running example of the geometric objects, this example modifies our `Circle` class.
- We now include a `setRadius` method in the `Circle` class that throws an `IllegalArgumentException` if the argument sent to `newRadius` is negative.

CircleWithException.java | GeometricObject.java | GeometricObject.java | TestCircleWithExcept

Exception Handling/src/GeometricObjec

```java
/* Circle Class - Classes in Java
 * Extends GeometricObject class - used in inheritance example
 *
 * MJL June 27, 2011
 * No known bugs
*/

public class CircleWithException extends GeometricObject {
   private double radius;
   private static int numberOfObjects = 0;

   /* default constructor */
   public CircleWithException() {
      //uncomment following line for constructor chaining display
      //System.out.println("In default Circle constructor.");
      this(1.0);
      numberOfObjects++;
   }//end default constructor

   /* radius specific constructor */
   public CircleWithException(double radius) {
      //uncomment following line for constructor chaining display
      //System.out.println("In radius specific Circle constructor.");
      setRadius(radius);
      numberOfObjects++;
   }//end radius specific constructor

   /* Return radius */
   public double getRadius() {
      return radius;
   }//end getRadius method
```

New addition to class

New addition to class

CircleWithException. | GeometricObject.java | GeometricObject.java | TestCircleWithExcept

```java
/* Set a new radius */
public void setRadius(double radius) throws IllegalArgumentException {
   if (radius >= 0)
       this.radius = radius;
   else throw new IllegalArgumentException(" Radius cannot be negative");
}//end setRadius method

/* Return area */
public double getArea() {
   return radius * radius * Math.PI;
}//end getArea method

/* Return diameter */
public double getDiameter() {
   return 2 * radius;
}//end getDiameter method

/* Return perimeter */
public double getPerimeter() {
   return 2 * radius * Math.PI;
}//end getPerimeter method

/* Return number of objects */
public static int getNumberOfObjects() {
     return numberOfObjects;
}//end getNumberOfObjects method
```

New addition to class

New addition to class

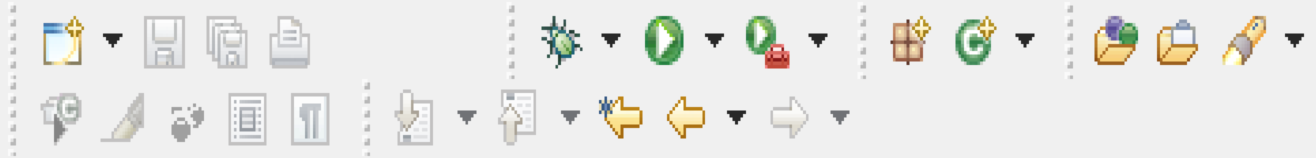
Test Class for `CircleWithException`

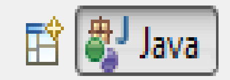
CircleWithException. | GeometricObject.java | TestCircleWithExcept

```
// Driver class to illustrate CircleWithExceptionClass
// MJL June 27, 2011

public class TestCircleWithException {
    public static void main(String args[]) {
        try{
            CircleWithException c1 = new CircleWithException(5);
            CircleWithException c2 = new CircleWithException(-5);
            CircleWithException c3 = new CircleWithException(0);
        }//end try block
        catch (IllegalArgumentException ex) {
            System.out.println(ex);
        }//end catch block

        System.out.println("Number of object created was: " +
                CircleWithException.getNumberOfObjects());
    }//end main method
}//end class TestCircleWithException
```

Java - Exception Handling/src/TestCircleWithException.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help





Console

```
<terminated> TestCircleWithException [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 27, 2011 1:33:5
In GeometricObject default constructor method
In GeometricObject default constructor method
java.lang.IllegalArgumentException:  Radius cannot be negative
Number of object created was: 1
```

Why did only 1 circle get created? Where is c3?

Answer: The exception occurred creating c2. Once the exception was handled no more statement in the try block are executed, so c3 was never built.

The `finally` Clause

- Occasionally, you may want some code to be executed regardless of whether an exception occurs or is caught.
- Java has a `finally` clause that can be used to accomplish this objective.
- The syntax for the finally clause looks like this:

```
try {
    statements;
}
catch (TheException ex) {
  //handling ex;
}
finally {
   finalStatements;
}
```

The `finally` Clause

- The code in the `finally` block is executed under all circumstances, regardless of whether an exception occurs in the `try` block or is caught.
- Consider three possible cases:

1. If no exception arises in the `try` block. `finalStatements` is executed, and the next statement after the `try` block is executed.
2. If one of the statements causes an exception in the `try` block that is caught in a `catch` block, the other statements in the `try` block are skipped, the `catch` block is executed, and the `finally` clause is executed. If the `catch` block does not re-throw an exception, the next statement after the `try` block is executed. If it does re-throw an exception, the exception is passed on to the caller of this method.
3. If one of the statements causes an exception that is not caught in any `catch` block, the other statements in the `try` block are skipped, the `finally` clause is executed, and the exception is passed to the caller of this method.

The `finally` Clause

- The `finally` block executes even if there is a `return` statement prior to reaching the finally block.
- The `catch` block may be omitted when the `finally` clause is used.
- A common use of the `finally` clause is in I/O programming. To ensure that a file is closed under all circumstances, you would place a file closing statement in the `finally` block. The example on the next page illustrates this use of the `finally` clause.

CircleWithException. | TestCircleWithExcept | FinallyDemo.java

```java
// Class to illustrate the use of the finally clause in a
// try/catch mechanism
// MJL June 27, 2011

public class FinallyDemo {
    public static void main(String args[]) {
        java.io.PrintWriter output = null;
        try {
            //create a file
            output = new java.io.PrintWriter("text.txt");
            //write formatted output to the file
            output.print("This is the first line in the file.\n");
            output.print("This is the second line in the file.\n");
            output.println("Welcome to file I/O in Java");
        }//end try block
        catch (java.io.IOException ex) {
            ex.printStackTrace();
        }//end catch block
        finally {
            //close the file in all cases
            if (output != null) output.close();
        }//end finally block
        System.out.println("Program terminates successfully!");
    }//end main method
}//end class FinallyDemo
```

Console

<terminated> FinallyDemo [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 27, 2011 1:59:20 PM)

```
Program terminates successfully!
```

Program output to console

Exception Handling

Name	Date modified	Type
.settings	6/23/2011 1:52 PM	File Fo
bin	6/27/2011 1:51 PM	File Fo
src	6/27/2011 1:51 PM	File Fo
.classpath	6/23/2011 1:52 PM	CLAS
.project	6/23/2011 1:52 PM	PROJE
text.txt	6/27/2011 1:59 PM	TXT F

text.txt Date modified: 6/27/2011 1:59 PM
TXT File Size: 102 bytes
Date created: 6/27/2011 1:56 PM

File created in current project workspace (by default).

C:\Users\Mark Llewellyn\eclipse\COP 3330 - Summer 2011\Excepti...

```
1 This is the first line in the file.
2 This is the second line in the file.
3 Welcome to file I/O in Java
4
```

Contents of the file.

When To Use Exception Handling

- The `try` block contains the code that is executed in normal circumstances.
- The `catch` block contains the code that is executed in exceptional circumstances.
- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.
- Be aware, however, that exception handling usually requires more time and resources, because it requires instantiating a new exception object, rolling back the call stack, and propagating the exception through the chain of methods invoked to search for the handler.
- An exception occurs in a method. If you want the exception to be processed by its caller, you should create an exception object and throw it. If you handle the exception in the method where it occurs, there is no need to throw or use exceptions.

When To Use Exception Handling

- In general, common exceptions that may occur in multiple classes in a project are candidates for exception classes.
- Simple errors that may occur in individual methods are best handled locally without throwing exceptions.
- When should you use `try-catch` block in the code?

Improper use of try-catch block, This code should be replace by:

```
try {
   System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```

```
if (refVar != null)
   System.out.println(refVar.toString());
else
   System.out.println("refVar is null");
}
```

Rethrowing Exceptions

- Java allows an exception handler to re-throw the exception if the handler cannot process the exception or simply wants its caller to be notified of the exception.
- The syntax may look like this:

```
try {
   statements;
}
catch (TheException ex) {
   //perform operations before exit;
   throw ex;
}
```

Chained Exceptions

- Sometimes a `catch` block will re-throw the original exception and sometimes you may need to throw a new exception (with additional information) along with the original exception.
- This is called chained exceptions.
- The example on the following page illustrates chained exceptions.

TestCircleWithExcept | FinallyDemo.java | *ChainedExceptions.j

```java
// Class to illustrate chained exceptions in Java
// MJL June 27, 2011

public class ChainedExceptions {
    public static void main(String args[]) {
        try {
            method1();
        }//end try block
        catch (Exception ex) {
            ex.printStackTrace();
        }//end catch block
    }//end main method

    public static void method1() throws Exception {
        try {
            method2();
        }//end try block
        catch (Exception ex) {
            throw new Exception("New exception info from method1()", ex);
        }//end catch block
    }//end method method1

    public static void method2() throws Exception {
        throw new Exception ("New exception infor from method2()");
    }//end method method2
}//end class ChainedExceptions
```

Console

<terminated> ChainedExceptions [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 27, 2011 2:23:02 PM)

```
java.lang.Exception: New exception info from method1()
	at ChainedExceptions.method1(ChainedExceptions.java:19)
	at ChainedExceptions.main(ChainedExceptions.java:7)
Caused by: java.lang.Exception: New exception info from method2()
	at ChainedExceptions.method2(ChainedExceptions.java:24)
	at ChainedExceptions.method1(ChainedExceptions.java:16)
	... 1 more
```

The main method invokes method1() and method1() invokes method2(). Method2() throws an exception. The exception thrown by method2() is caught in the catch block of method1() and is wrapped in a new exception. The new exception created by method1() is thrown and caught in the catch block of the main method, which prints the stack trace. So you see the new exception thrown by method1() first, followed by the exception thrown by method2().