IT2205: Programming I

Section 7

Exception Handling (05 hrs)

Exception Handling-Fundamentals

- An exception is an abnormal condition that arises in a code sequence at run time
- A Java exception is an object that describes an exceptional condition that has occurred in a piece of code
- When an exceptional condition arises, an object representing that exception is created and thrown in the method that caused the error
- An exception can be caught to handle it or pass it on
- Exceptions can be generated by the Java run-time system, or they can be manually generated by your code

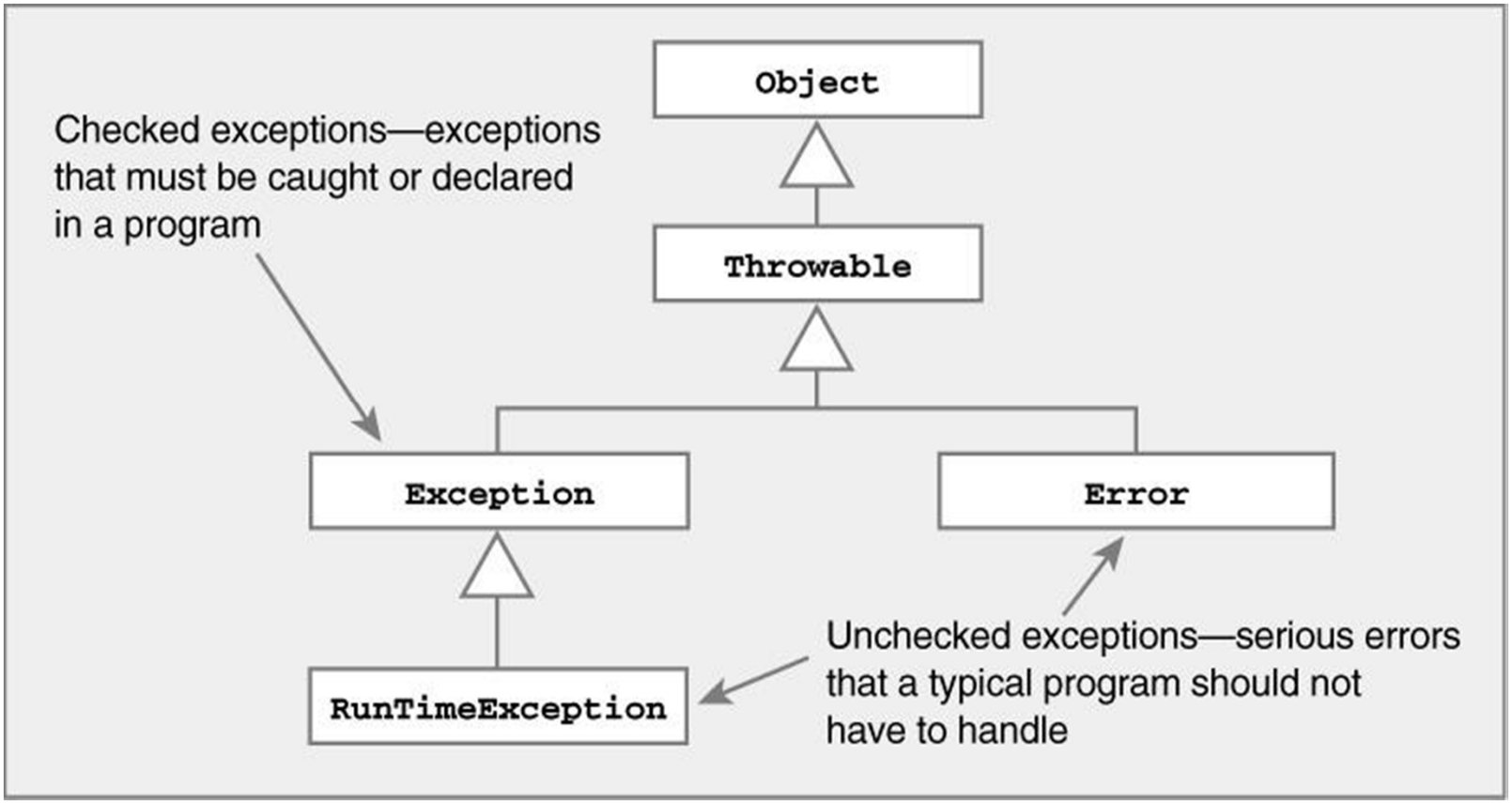
Exception Handling

- Performing action in response to exception

Examples

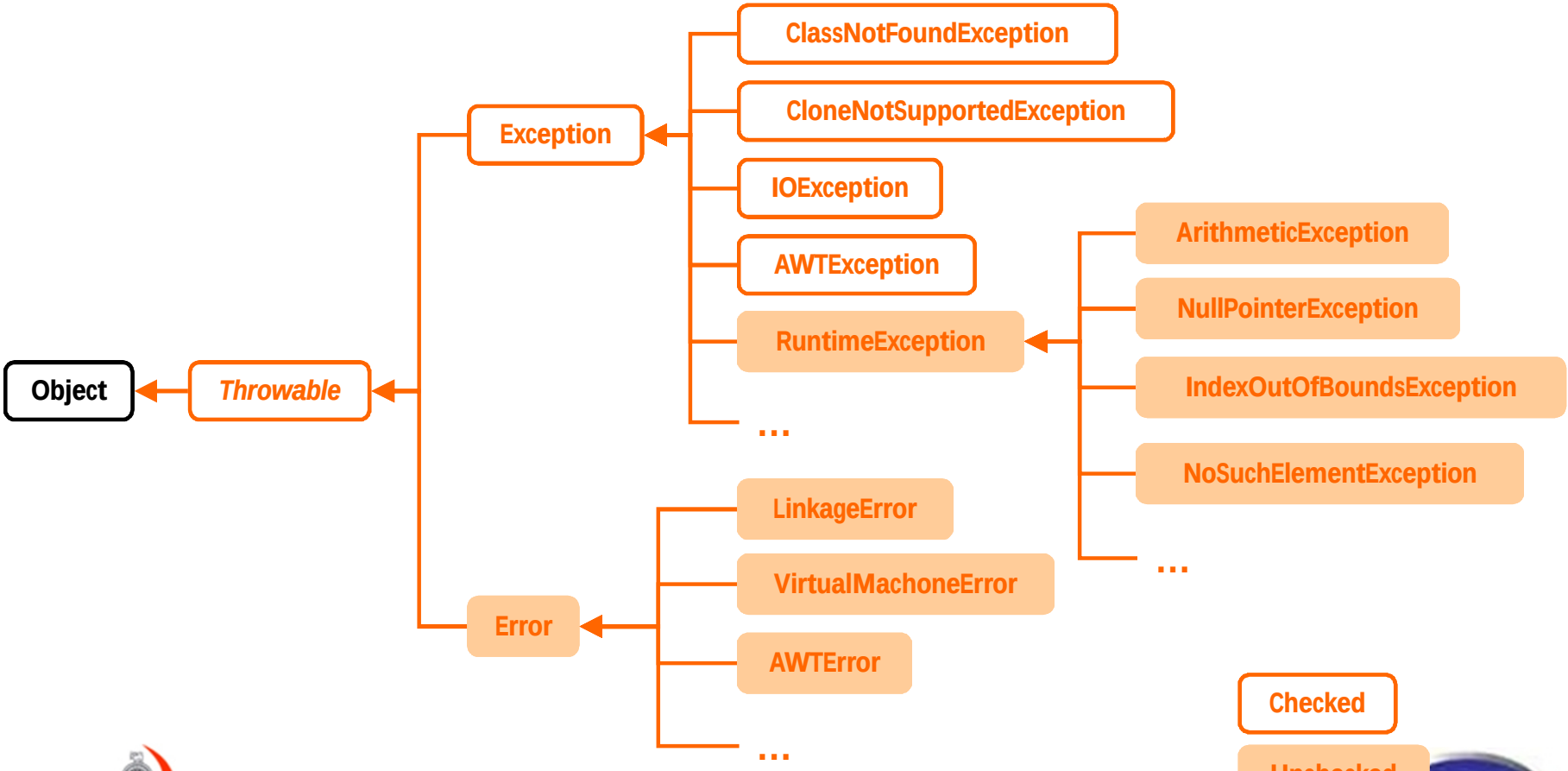
- Exit program (abort)
- Deal with exception and continue
  - Print error message
  - Request new data
  - Retry action

Representing Exceptions



Representing Exceptions

- Java Exception class hierarchy



Exception Handling in Java

- Java exception handling is managed by via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**
- Program statements to monitor are contained within a **try** block
- If an exception occurs within the **try** block, it is thrown
- Code within **catch** block catch the exception and handle it

Example

```
class Exc2 {
  public static void main(String args[]) {
    int d, a;

    try { // monitor a block of code.
      d = 0;
      a = 42 / d;
      System.out.println("This will not be printed.");
    } catch (ArithmeticException e) { // catch divide-by-zero error
      System.out.println("Division by zero.");
    }
    System.out.println("After catch statement.");
  }
}
```

Output:

Division by zero.

After catch statement.

try and catch statement

- The scope of a **catch** clause is restricted to those statements specified by the immediately preceding **try** statement.
- A **catch** statement cannot catch an exception thrown by another **try** statement.
- The statements that are protected by the **try** must be surrounded by curly braces.

Multiple Catch Clauses

- If more than one can occur, then we use multiple catch clauses
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed
- After one **catch** statement executes, the others are bypassed

Example

```
class MultiCatch {
  public static void main(String args[]) {
    try {
      int a = args.length;
      System.out.println("a = " + a);
      int b = 42 / a;
      int c[] = { 1 };
      c[42] = 99;
    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    } catch(ArrayIndexOutOfBoundsException e) {
      System.out.println("Array index oob: " + e);
    }
    System.out.println("After try/catch blocks.");
  }
}
```

Caution

- Exception subclass must come before any of of their superclasses
- A **catch** statement that uses a superclass will catch exceptions of that type plus any of its subclasses. So, the subclass would never be reached if it come after its superclass
- For example, **ArithmeticException** is a subclass of **Exception**
- Moreover, unreachable code in Java generates error

Example

```
/*  This program contains an error.

    A subclass must come before its superclass in
    a series of catch statements. If not,
    unreachable code will be created and a
    compile-time error will result.
*/
class SuperSubCatch {
  public static void main(String args[]) {
    try {
      int a = 0;
      int b = 42 / a;
    } catch(Exception e) {
      System.out.println("Generic Exception catch.");
    }
    /* This catch is never reached because
       ArithmeticException is a subclass of Exception. */
    catch(ArithmeticException e) { // ERROR - unreachable
      System.out.println("This is never reached.");
    }
  }
}
```

Nested try Statements

- A **try** statement can be inside the block of another try
- Each time a **try** statement is entered, the context of that exception is pushed on the stack
- If an inner **try** statement does not have a catch, then the next **try** statement's catch handlers are inspected for a match
- If a method call within a **try** block has **try** block within it, then then it is still nested **try**

Example

```
// An example nested try statements.
class NestTry {
  public static void main(String args[]) {
    try {
      int a = args.length;

      /* If no command line args are present,
         the following statement will generate
         a divide-by-zero exception. */
      int b = 42 / a;

      System.out.println("a = " + a);

      try { // nested try block
        /* If one command line arg is used,
           then an divide-by-zero exception
           will be generated by the following code. */
        if(a==1) a = a/(a-a); // division by zero

        /* If two command line args are used
           then generate an out-of-bounds exception. */
        if(a==2) {
          int c[] = { 1 };
          c[42] = 99; // generate an out-of-bounds exception
        }
      } catch(ArrayIndexOutOfBoundsException e) {
        System.out.println("Array index out-of-bounds: " + e);
      }

    } catch(ArithmeticException e) {
      System.out.println("Divide by 0: " + e);
    }
  }
}
```

throw

- It is possible for your program to to throw an exception explicitly

  throw *ThrowableInstance*

- Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass **Throwable**
- There are two ways to obtain a **Throwable** objects:
  - Using a parameter into a catch clause
  - Creating one with the **new** operator

Example -throw Statements

```
// Demonstrate throw.
class ThrowDemo {
  static void demoproc() {
    try {
      throw new NullPointerException("demo");
    } catch(NullPointerException e) {
      System.out.println("Caught inside demoproc.");
      throw e; // re-throw the exception
    }
  }

  public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      System.out.println("Recaught: " + e);
    }
  }
}
```

Output:

Caught inside demoproc.

Recaught: java.lang.NullPointerException: demo

throws

- If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception

- *type method-name parameter-list)* throws *exception-list*

```
{
    // body of method
}
```

- It is not applicable for **Error** or **RuntimeException,** or any of their subclasses

Example: incorrect program

```
// This program contains an error and will not compile.
class ThrowsDemo {
  static void throwOne() {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    throwOne();
  }
}
```

Example: corrected version

```
// This is now correct.
class ThrowsDemo {
  static void throwOne() throws IllegalAccessException {
    System.out.println("Inside throwOne.");
    throw new IllegalAccessException("demo");
  }
  public static void main(String args[]) {
    try {
      throwOne();
    } catch (IllegalAccessException e) {
      System.out.println("Caught " + e);
    }
  }
}
```

Output:

Inside throwOne.

Caught java.lang.IllegalAccessException: demo

Finally Statement

- **finally** creates a block of code that will be executed after a **try/catch** block has completed and before the code following the **try/catch** block.
- **finally** block will be executed whether or not an exception is thrown.
- Any time a method is about to return to the caller from inside a try/catch block, via an uncaught exception or an explicit return statement, the finally clause is also executed just before the method returns.
- Each try clause requires at least one catch or finally clause.

Example

```
// Demonstrate finally.
class FinallyDemo {
   // Through an exception out of the method.
   static void procA() {
      try {
         System.out.println("inside procA");
         throw new RuntimeException("demo");
      } finally {
         System.out.println("procA's finally");
      }
   }

   // Return from within a try block.
   static void procB() {
      try {
         System.out.println("inside procB");
         return;
      } finally {
         System.out.println("procB's finally");
      }
   }

   // Execute a try block normally.
   static void procC() {
      try {
         System.out.println("inside procC");
      } finally {
         System.out.println("procC's finally");
      }
   }

   public static void main(String args[]) {
      try {
         procA();
      } catch (Exception e) {
         System.out.println("Exception caught");
      }
      procB();
      procC();
   }
}
```

Output

```
inside procA
procA's finally
Exception caught
inside procB
procB's finally
inside procC
procC's finally
```

Uncaught Exceptions

```
class exc0{
public static void main(String args[])
{
    int d=0;
    int a=42/d;
}
}
```

- A new exception object is constructed and then thrown.
- This exception is caught by the default handler provided by the java runtime system.
- The default handler displays a string describing the exception, prints the stack trace from the point at which the exception occurred and terminates the program.

Output:

```
java.lang.ArithmeticException: / by zero
        at exc0.main(exc0.java:4)
```

Displaying a Description of an Exception

- **Throwable** overrides the toString() method (defined by **Object**) so that it returns a string containing a description of the exception.
- **Example:**

```
catch(ArithmeticException e)
{
        System.out.println("Exception: "+e);
}
```

- **Output:**

Exception: java.lang.ArithmeticException: / by zero

User Defined Exception

- Define a subclass of the Exception class.
- The new subclass inherits all the methods of Exception and can override them.

```
class MyException extends Exception{
private int a;
MyException(int i) { a = i;}
public String toString (){ return “MyException[“ + a+”]”;}
}
```

Continuation of the Example

```
class test{
    static void compute (int a) throws Myexception{
            if(a>10) throw new MyException(a);
            System.out.println("Normal Exit");
    }
public static void main(String args[]){
    try{
            compute(1);
            compute(20);
            }catch(MyException e){ System.out.println("Caught " +e);
}
}
```

Example-2

```
class InvalidRadiusException extends Exception {
        private double r;
        public InvalidRadiusException(double radius){
            r = radius;
        }
        public void printError(){
            System.out.println("Radius [" +  r + "] is not valid");
        }
}
```

Continuation of Example-2

```
class Circle    {
        double x, y, r;

        public Circle (double centreX, double centreY, double radius ) throws
    InvalidRadiusException {
        if (r <= 0 ) {
                throw new InvalidRadiusException(radius);
        }
        else {
                x = centreX ; y = centreY;  r = radius;
        }
     }
}
```

Continuation of Example-2

```
class CircleTest {
      public static void main(String[] args){
       try{
              Circle c1 = new Circle(10, 10, -1);
              System.out.println("Circle created");
       }
       catch(InvalidRadiusException e)
       {
              e.printError();
       }
     }
}
```

Inside the standard package java.lang, Java defines several exception classes

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Java's Unchecked RuntimeException Subclasses Defined in java.lang

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions. (Added by JDK 7.)

Chained Exceptions

- Beginning with JDK 1.4, a feature was incorporated into the exception subsystem: *chained exceptions*. The chained exception feature allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an **ArithmeticException** because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. Although the method must certainly throw an **ArithmeticException**, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

Chained Exceptions cont...

Throwable(Throwable *causeExc*)

Throwable(String *msg*, Throwable *causeExc*)

- In the first form, *causeExc* is the exception that causes the current exception. That is, *causeExc* is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.
- The chained exception methods added to **Throwable** are **getCause( )** and **initCause( )**.

Throwable getCause( )

Throwable initCause(Throwable *causeExc*)

Chained Exceptions cont...

- The **getCause( )** method returns the exception that underlies the current exception. If there is no underlying exception, **null** is returned. The **initCause( )** method associates *causeExc* with the invoking exception and returns a reference to the exception. Thus, you can associate a cause with an exception after the exception has been created. However, the cause exception can be set only once. Thus, you can call **initCause( )** only once for each exception object. Furthermore, if the cause exception was set by a constructor, then you can't set it again using **initCause( )**. In general, **initCause( )** is used to set a cause for legacy exception classes that don't support the two additional constructors described earlier.

Chained Exceptions - Example

```
// Demonstrate exception chaining.
class ChainExcDemo {
static void demoproc() {

   // create an exception
   NullPointerException e =
     new NullPointerException("top layer");

   // add a cause
   e.initCause(new ArithmeticException("cause"));

   throw e;
 }
```

Chained Exceptions – Example cont...

```
public static void main(String args[]) {
    try {
      demoproc();
    } catch(NullPointerException e) {
      // display top level exception
      System.out.println("Caught: " + e);

      // display cause exception
      System.out.println("Original cause: " +
                         e.getCause());
    }
  }
}
```

Three New JDK 7 Exception Features

- JDK 7 adds three interesting and useful features to the exception system.
- The first automates the process of releasing a resource, such as a file, when it is no longer needed. It is based on an expanded form of the **try** statement called ***try****-with-resources*,
- The second new feature is called *multi-catch*, and the third is sometimes referred to as *final rethrow* or *more precise rethrow*.

Three New JDK 7 Exception Features cont...

- The multi-catch feature allows two or more exceptions to be caught by the same **catch** clause.
- It is not uncommon for two or more exception handlers to use the same code sequence even though they respond to different exceptions.
- Instead of having to catch each exception type individually, now you can use a single **catch** clause to handle all of the exceptions without code duplication.

Three New JDK 7 Exception Features cont...

- To use a multi-catch, separate each exception type in the **catch** clause with the OR operator.
- Each multi-catch parameter is implicitly **final**. (You can explicitly specify **final**, if desired, but it is not necessary.)
- Because each multi-catch parameter is implicitly **final**, it can't be assigned a new value.

Three New JDK 7 Exception Features cont...

- Here is a catch statement that uses the multi-catch feature to catch both ArithmeticException and ArrayIndexOutOfBoundsException:

```
catch(ArithmeticException | ArrayIndexOutOfBoundsException
e) {
```

Three New JDK 7 Exception Features cont...

```
// Demonstrate JDK 7's multi-catch feature.
class MultiCatch {
  public static void main(String args[]) {
    int a=10, b=0;
    int vals[] = { 1, 2, 3 };
    try {
      int result = a / b; // generate an ArithmeticException

//      vals[10] = 19; // generate an ArrayIndexOutOfBoundsException

      // This catch clause catches both exceptions.
    } catch(ArithmeticException | ArrayIndexOutOfBoundsException e) {
      System.out.println("Exception caught: " + e);
    }
    System.out.println("After multi-catch.");
  }
}
```

Three New JDK 7 Exception Features cont...

- The program will generate an **ArithmeticException** when the division by zero is attempted. If you comment out the division statement and remove the comment symbol from the next line, an **ArrayIndexOutOfBoundsException** is generated. Both exceptions are caught by the single **catch** statement.

Three New JDK 7 Exception Features cont...

- The more precise rethrow feature restricts the type of exceptions that can be rethrown to only those checked exceptions that the associated **try** block throws, that are not handled by a preceding **catch** clause, and that are a subtype or supertype of the parameter.
- Although this capability might not be needed often, it is now available for use. For the more precise rethrow feature to be in force, the **catch** parameter must be either effectively **final**, which means that it must not be assigned a new value inside the **catch** block, or explicitly declared **final**.

The End of Section