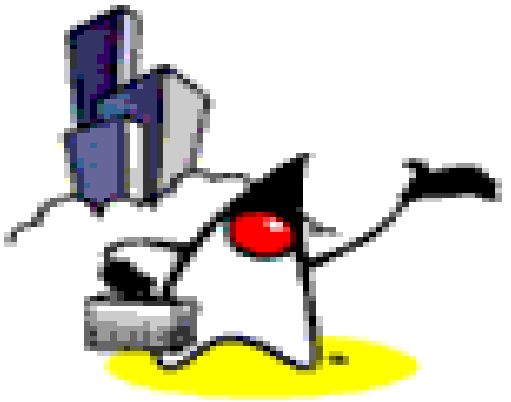
Java Exception Handling

(Importance scale: ****, Very important)

Sang Shin
Michèle Garoche
www.javapassion.com
"Learn with Passion!"

Topics

- What is an Exception?
- What happens when an Exception occurs?
- Benefits of Exception Handling framework
- Catching exceptions with *try-catch*
- Catching exceptions with *finally*
- Throwing exceptions
- Rules in exception handling
- Exception class hierarchy
- Checked exception and unchecked exception
- Creating your own exception class
- Assertions



What is an Exception?

What is an Exception?

- Represents an exceptional event - typically an error that occurs during run-time
- Causes normal program flow to be disrupted
- Exception examples
  - Divide by zero error
  - Accessing the elements of an array beyond its range
  - Invalid input
  - Hard disk crash
  - Opening a non-existent file
  - Heap memory exhausted

Exception Causing Code Example

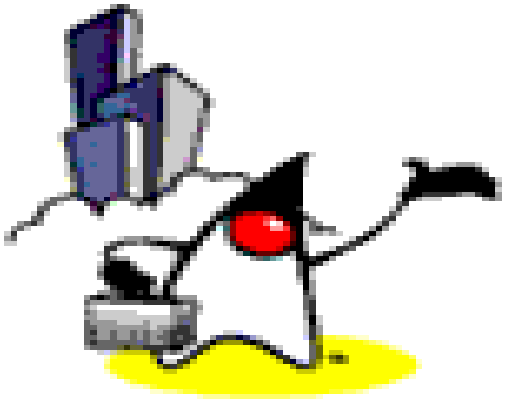
```
1 class DivByZero {
2     public static void main(String args[]) {
3         // The code below will cause an
4         // Exception to occur
5         System.out.println(3/0);
6         System.out.println("Pls. print me.");
7     }
8 }
```

Example: Default Exception Handling by Java Runtime

- Displays this error message

```
Exception in thread "main"
  java.lang.ArithmeticException: / by zero
       at DivByZero.main(DivByZero.java:3)
```

- Default exception handler
  - Provided by Java runtime
  - Prints out exception description
  - Prints the stack trace
    - Chain of methods where the exception occurred
  - Causes the program to terminate



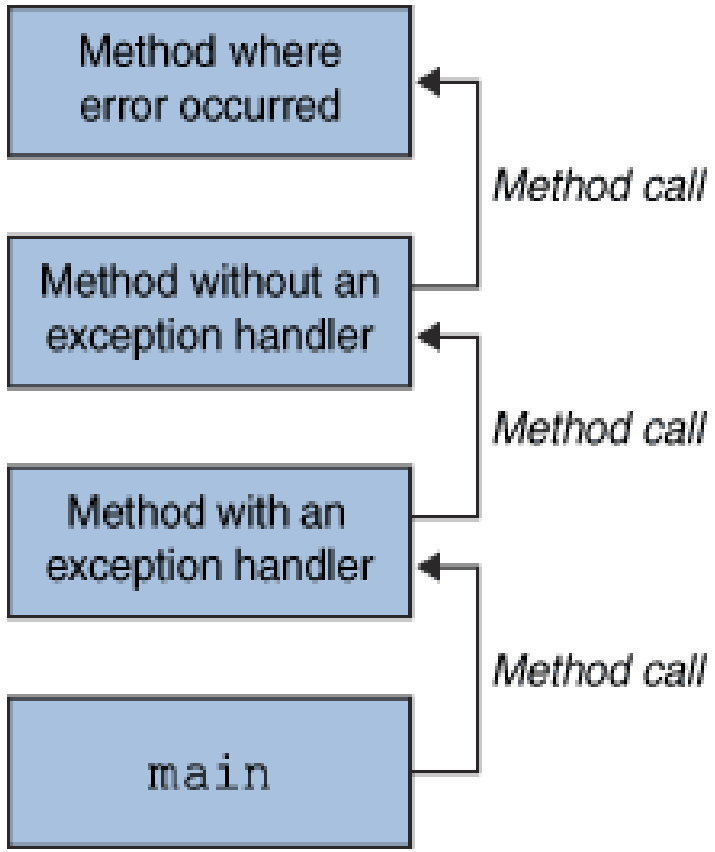
What Happens When an Exception Occurs?

What Happens When an Exception Occurs?

- When an exception occurs within a method, the method typically creates an *Exception* object and hands it off to the Java runtime system
- Creating an exception object and handing it to the runtime system is called “throwing an exception”
- *Exception* object contains information about the error, including exception type and the state of the program when the error occurred

What Happens When an Exception Occurs?

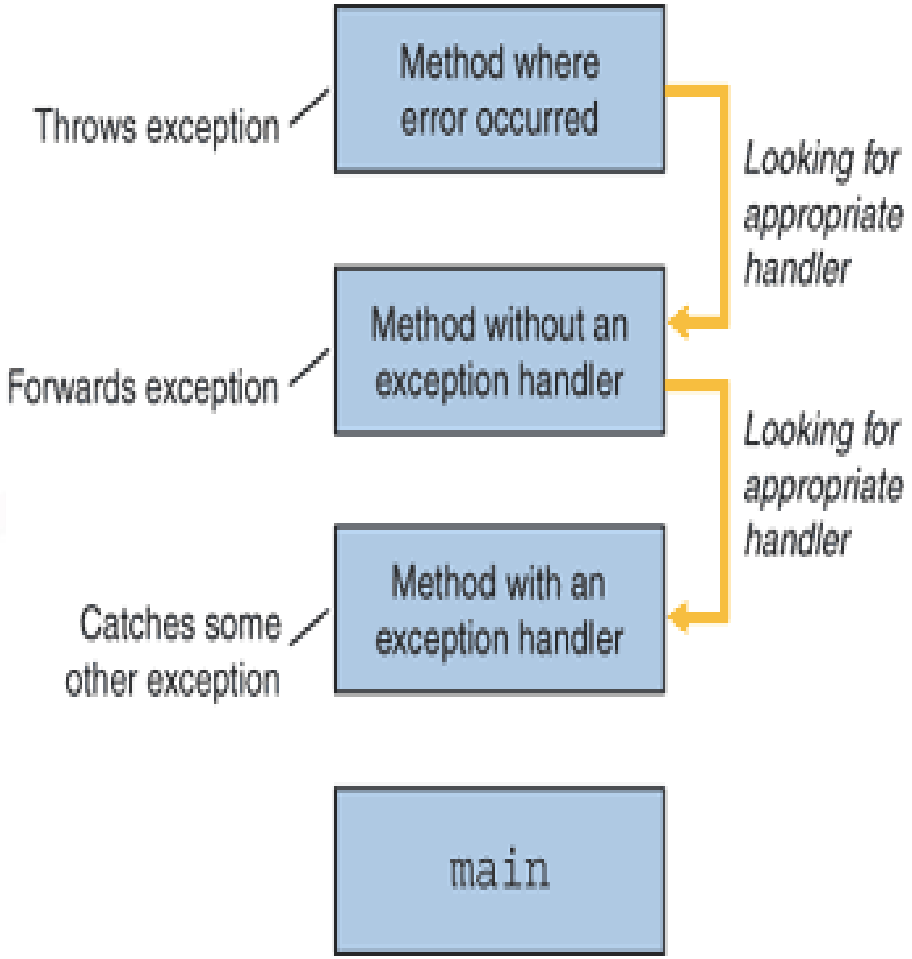
- The Java runtime system searches the call stack for a method that contains an exception handler

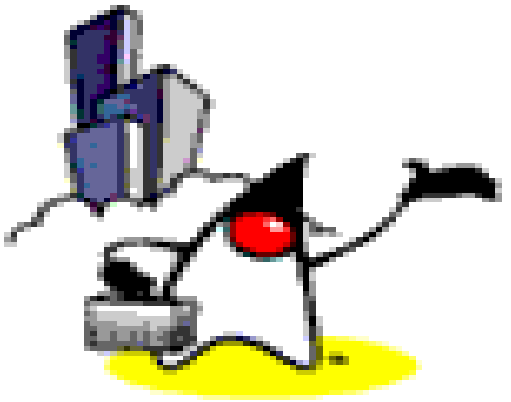


What Happens When an Exception Occurs? (Continued)

- When an appropriate handler is found, the runtime system passes the exception to the handler
  - An exception handler is considered “appropriate” if the type of the exception object thrown matches the type that can be handled by the exception handler
  - The exception handler chosen is said to “catch the exception”.
- If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates and uses the default exception handler

Searching the Call Stack for an Exception Handler





Benefits of Exception Handling Framework

Benefits of Java Exception Handling Framework

- #1: Separating Error-Handling code from “regular” business logic code
  - Cleaner code since error handling code and business logic code do not have to be mixed up
- #2: Whoever best suited to handle the error will handle the error
  - Errors will be propagated up the call stack until an “appropriate” handler is found
- #3: Grouping and differentiating errors based on their types
  - Exception classes are genuine Java classes
  - Exception classes have inheritance hierarchy among themselves

#1: Separating Error Handling Code from Regular Code

- In traditional programming, error detection, reporting, and handling often lead to confusing spaghetti code
- Consider pseudocode method here that reads an entire file into memory

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

#1: Traditional Programming: No separation of error handling code

- In traditional programming, To handle such cases, the *readFile* function must have more code to do error detection, reporting, and handling.

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
```

#1: Traditional Programming: No separation of error handling code

```
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

#1: Separating Error Handling Code from Regular Code (in Java)

- Java Exception framework enable you to write the main flow of your code and to deal with the errors elsewhere

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

#1: Separating Error Handling Code from Regular Code

- Note that exceptions don't spare you the effort of doing the work of detecting, reporting, and handling errors, but they do help you organize the work more effectively.

#2: Propagating Errors Up the Call Stack

- Suppose that the *readFile* method is the fourth method in a series of nested method calls made by the main program: method1 calls *method2*, which calls *method3*, which finally calls *readFile*
- Suppose also that *method1* is the only method interested in the errors that might occur within *readFile*.

```
method1 {
    call method2;
}

method2 {
    call method3;
}

method3 {
    call readFile;
}
```

#2: Traditional Way of Propagating Errors

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

- Traditional error-notification techniques force method2 and method3 to propagate the error codes returned by readFile up the call stack until the error codes finally reach method1—the only method that is interested in them.

#2: Propagating Errors in Java

```
method1 {
   try {
      call method2;
   } catch (exception e) {
      doErrorProcessing;
   }
}

method2 throws exception {
   call method3;
}

method3 throws exception {
   call readFile;
}
```

- A method can duck any exceptions thrown within it, thereby allowing a method farther up the call stack to catch it. Hence, only the methods that care about errors have to worry about detecting errors
- Any checked exceptions that can be thrown within a method must be specified in its throws clause.

#3: Grouping and Differentiating Error Types

- Because all exceptions thrown within a program are genuine objects, the grouping or categorizing of exceptions is a natural outcome of the class hierarchy
- An example of a group of related exception classes in the Java platform are those defined in *java.io* — *IOException* and its descendants
  - *IOException* is the most general and represents any type of error that can occur when performing I/O
  - Its descendants represent more specific errors. For example, *FileNotFoundException* means that a file could not be located on disk.

#3: Grouping and Differentiating Error Types

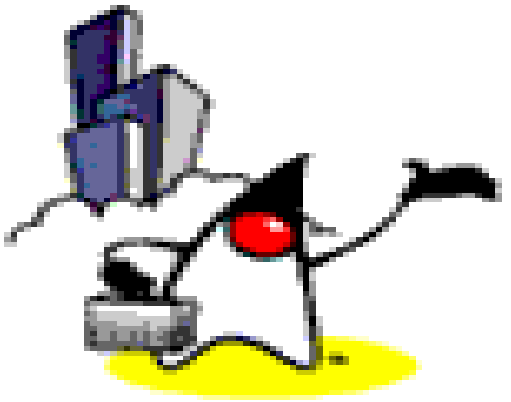
- A method can write specific handlers that can handle a very specific exception
- The *FileNotFoundException* class has no descendants, so the following handler can handle only one type of exception.

```
catch (FileNotFoundException e) {
    ...
}
```

#3: Catching Errors with Super Type

- A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement. For example, to catch all I/O exceptions, regardless of their specific type, an exception handler specifies an *IOException* argument.

```
// Catch all I/O exceptions, including
// FileNotFoundException, EOFException, and so on.
catch (IOException e) {
    ...
}
```



Catching Exceptions with try-catch

Catching Exceptions: The *try-catch* Statements

- Syntax:

```
try {
   <code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
   <handler if ExceptionType1 occurs>
}
...
} catch (<ExceptionTypeN> <ObjName>) {
   <handler if ExceptionTypeN occurs>
}
```

Catching Exceptions: The *try-catch* Statements

```
1  class DivByZero {
2     public static void main(String args[]) {
3        try {
4           System.out.println(3/0);
5           System.out.println("Please print me.");
6        } catch (ArithmeticException exc) {
7           //Division by zero is an ArithmeticException
8           System.out.println(exc);
9        }
10       System.out.println("After exception.");
11    }
12 }
```

Catching Exceptions: Multiple catch

```
1  class MultipleCatch {
2     public static void main(String args[]) {
3        try {
4           int den = Integer.parseInt(args[0]);
5           System.out.println(3/den);
6        } catch (ArithmeticException exc) {
7           System.out.println("Divisor was 0.");
8        } catch (ArrayIndexOutOfBoundsException exc2) {
9           System.out.println("Missing argument.");
10       }
11       System.out.println("After exception.");
12    }
13 }
```

Catching Exceptions: Nested try's

```
class NestedTryDemo {
    public static void main(String args[]){
        try {
            int a = Integer.parseInt(args[0]);
            try {
                int b = Integer.parseInt(args[1]);
                System.out.println(a/b);
            } catch (ArithmeticException e) {
                System.out.println("Div by zero error!");
            }
            //continued...
```

Catching Exceptions: Nested try's

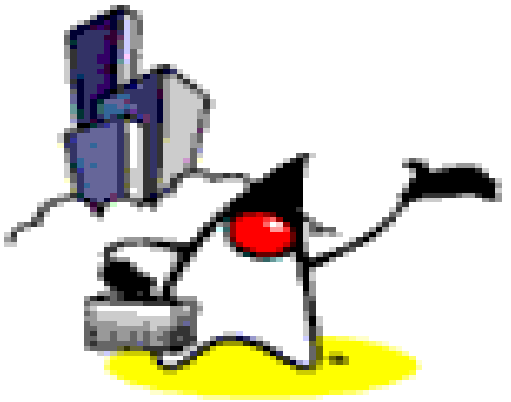
```
        } catch (ArrayIndexOutOfBoundsException) {
            System.out.println("Need 2 parameters!");
        }
    }
}
```

Catching Exceptions: Nested try's with methods

```
1  class NestedTryDemo2 {
2     static void nestedTry(String args[]) {
3        try {
4           int a = Integer.parseInt(args[0]);
5           int b = Integer.parseInt(args[1]);
6           System.out.println(a/b);
7        } catch (ArithmeticException e) {
8           System.out.println("Div by zero error!");
9        }
10    }
11 //continued...
```

Catching Exceptions: Nested try's with methods

```
12      public static void main(String args[]){
13          try {
14              nestedTry(args);
15          } catch (ArrayIndexOutOfBoundsException e) {
16              System.out.println("Need 2 parameters!");
17          }
18      }
19  }
```



Catching Exceptions with finally

Catching Exceptions: The *finally* Keyword

- Syntax:

```
try {
    <code to be monitored for exceptions>
} catch (<ExceptionType1> <ObjName>) {
    <handler if ExceptionType1 occurs>
} ...
} finally {
    <code to be executed before the try block ends>
}
```

- Contains the code for cleaning up after a try or a catch

Catching Exceptions: The *finally* Keyword

- Block of code is always executed despite of different scenarios:
  - Forced exit occurs using a *return*, a *continue* or a *break* statement
  - Normal completion
  - Caught exception thrown
    - Exception was thrown and caught in the method
  - Uncaught exception thrown
    - Exception thrown was not specified in any catch block in the method

Catching Exceptions: The *finally* Keyword

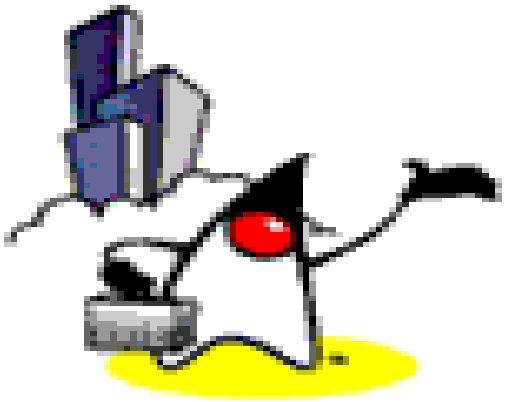
```
1  class FinallyDemo {
2     static void myMethod(int n) throws Exception{
3        try {
4           switch(n) {
5              case 1: System.out.println("1st case");
6                      return;
7              case 3: System.out.println("3rd case");
8                      throw new RuntimeException("3!");
9              case 4: System.out.println("4th case");
10                     throw new Exception("4!");
11             case 2: System.out.println("2nd case");
12          }
13 //continued...
```

Catching Exceptions: The *finally* Keyword (Continued)

```
14          } catch (RuntimeException e) {
15                  System.out.print("RuntimeException: ");
16                  System.out.println(e.getMessage());
17          } finally {
18                  System.out.println("try-block entered.");
19          }
20      }
21  //continued...
```

Catching Exceptions: The *finally* Keyword (Continued)

```
22      public static void main(String args[]){
23          for (int i=1; i<=4; i++) {
24              try {
25                  FinallyDemo.myMethod(i);
26              } catch (Exception e){
27                  System.out.print("Exception caught: ");
28                  System.out.println(e.getMessage());
29              }
30              System.out.println();
31          }
32      }
33  }
```



Throwing Exceptions

Throwing Exceptions: The *throw* Keyword

- Java allows you to throw exceptions (generate exceptions) using *throw* keyword

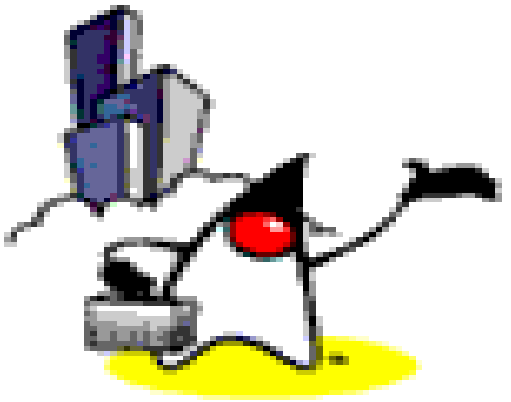
```
throw <exception object>;
```

- An exception you throw is an object
  - You have to create an exception object in the same way you create any other object
- Example:

```
throw new ArithmeticException("testing...");
```

Example: Throwing Exceptions

```
1  class ThrowDemo {
2     public static void main(String args[]){
3        String input = "invalid input";
4        try {
5           if (input.equals("invalid input")) {
6              throw new RuntimeException("throw demo");
7           } else {
8              System.out.println(input);
9           }
10          System.out.println("After throwing");
11       } catch (RuntimeException e) {
12          System.out.println("Exception caught:" + e);
13       }
14    }
15 }
```



Rules in Exception Handling

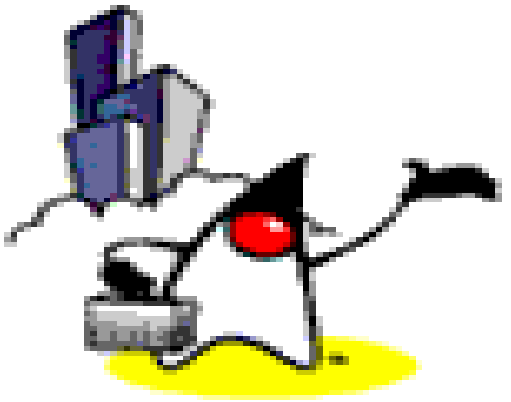
Rules on Exceptions

- A method is required to either catch or list all exceptions it might throw
  - Except for *Error* or *RuntimeException*, or their subclasses
- If a method may cause an exception to occur but does not catch it, then it must say so using the *throws* keyword
  - Applies to checked exceptions only
- Syntax:

```
<type> <methodName> (<parameterList>)
  throws <exceptionList> {
    <methodBody>
}
```

Example: Method throwing an Exception

```
1  class ThrowingClass {
2     static void meth() throws ClassNotFoundException {
3        throw new ClassNotFoundException ("demo");
4     }
5  }
6  class ThrowsDemo {
7     public static void main(String args[]) {
8        try {
9           ThrowingClass.meth();
10       } catch (ClassNotFoundException e) {
11          System.out.println(e);
12       }
13    }
14 }
```



Exception Class Hierarchy

The Error and Exception Classes

- Throwable class
  - Root class of exception classes
  - Immediate subclasses
    - *Error*
    - *Exception*
- Exception class
  - Conditions that user programs can reasonably deal with
  - Usually the result of some flaws in the user program code
  - Examples
    - Division by zero error
    - Array out-of-bounds error

The Error and Exception Classes

- Error class
  - Used by the Java run-time system to handle errors occurring in the run-time environment
  - Generally beyond the control of user programs
  - Examples
    - Out of memory errors
    - Hard disk crash

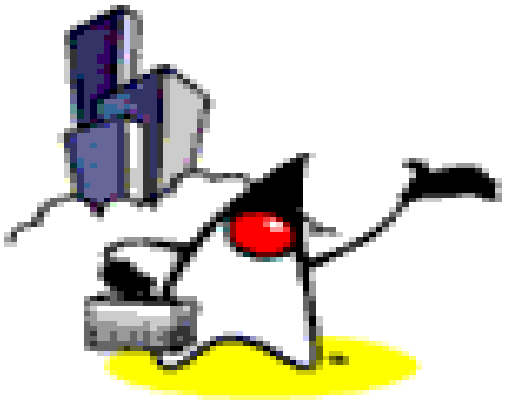
Exception Classes and Hierarchy

Exception Class Hierarchy			
Throwable	Error	LinkageError, ...	
		VirtualMachineError, ...	
	Exception	ClassNotFoundException,	
		CloneNotSupportedException,	
		IllegalAccessException,	
		InstantiationException,	
		InterruptedException,	
		IOException,	EOFException,
			FileNotFoundException,
			...
		RuntimeException,	ArithmeticException,
			ArrayStoreException,
			ClassCastException,
			IllegalArgumentException, (IllegalThreadStateException and NumberFormatException as subclasses)
			IllegalMonitorStateException,
			IndexOutOfBoundsException,
			NegativeArraySizeException,
			NullPointerException,
			SecurityException
		...	

Exception Classes and Hierarchy

- Multiple catches should be ordered from subclass to superclass.

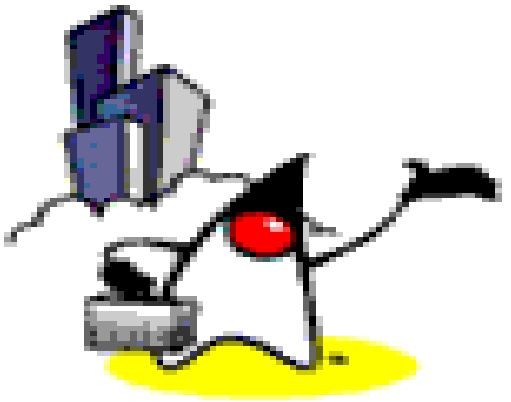
```
1  class MultipleCatchError {
2     public static void main(String args[]){
3         try {
4             int a = Integer.parseInt(args [0]);
5             int b = Integer.parseInt(args [1]);
6             System.out.println(a/b);
  -         } catch (ArrayIndexOutOfBoundsException e) {
  -         } catch (Exception ex) {
1         }
2     }
3  }
```



Checked Exceptions & Unchecked Exceptions

Checked and Unchecked Exceptions

- Checked exception
  - Java compiler checks if the program either catches or lists the occurring checked exception
  - If not, compiler error will occur
- Unchecked exceptions
  - Not subject to compile-time checking for exception handling
  - Built-in unchecked exception classes
    - Error
    - RuntimeException
    - Their subclasses
  - Handling all these exceptions may make the program cluttered and may become a nuisance



Creating Your Own Exception Class

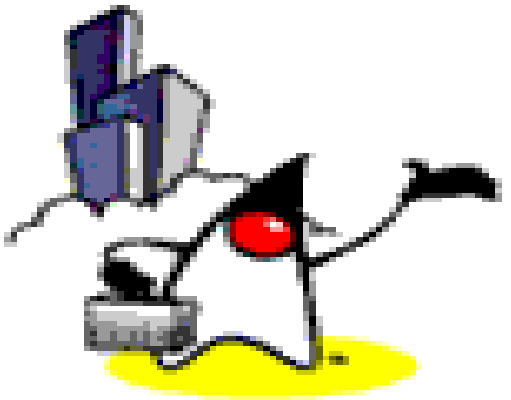
Creating Your Own Exception Class

- Steps to follow
  - Create a class that *extends* the *RuntimeException* or the *Exception* class
  - Customize the class
    - Members and constructors may be added to the class
- Example:

```
1 class HateStringExp extends RuntimeException {
2     /* some code */
3 }
```

How To Use Your Own Exceptions

```
1  class TestHateString {
2      public static void main(String args[]) {
3          String input = "invalid input";
4              try {
5                  if (input.equals("invalid input")) {
6                      throw new HateStringExp();
7                  }
8                  System.out.println("Accept string.");
9              } catch (HateStringExp e) {
10                 System.out.println("Hate string!”);
11             }
12     }
13 }
```



Assertions

What are Assertions?

- Allow the programmer to find out if the program behaves as expected
- Informs the person reading the code that a particular condition should always be satisfied
  - Running the program informs you if assertions made are true or not
  - If an assertion is not true, an *AssertionError* is thrown
- User has the option to turn it off or on when running the application

Enabling or Disabling Assertions

- Program with assertions may not work properly if used by clients not aware that assertions were used in the code

- Enabling assertions:
  - Use the –enableassertions or –ea switch.

```
java -enableassertions MyProgram
```

Assert Syntax

- Two forms:
  - Simpler form:

    ```
    assert <expression1>;
    ```

    where
    - <expression1> is the condition asserted to be true
  - Other form:

    ```
    assert <expression1> : <expression2>;
    ```

    where
    - <expression1> is the condition asserted to be true
    - <expression2> is some information helpful in diagnosing why the statement failed

Assert Syntax

```
1  class AgeAssert {
2     public static void main(String args[]) {
3        int age = Integer.parseInt(args[0]);
4        assert(age>0);
5        /* if age is valid (i.e., age>0) */
6        if (age >= 18) {
7           System.out.println("You're an adult! =)");
8        }
9     }
10 }
```

Thank you!

Check JavaPassion.com Codecamps!
http://www.javapassion.com/codecamps
"Learn with Passion!"