Exception Hnadling

- 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?

- Exception – an exceptional event which is an indication of a problem that occurs during a program’s execution
- Examples
  - Divide by zero errors
  - Accessing the elements of an array beyond its range
  - Invalid input
  - Hard disk crash
  - Opening a non-existent file
  - Heap memory exhausted

Exception Example

```
class DivByZero {
   public static void main(String args[]) {
      System.out.println(3/0);
      System.out.println(“Pls. print me.”);}
   }
```

Exception Examples

- `ArrayIndexOutOfBoundsException` – an attempt is made to access an element past the end of an array
- `ClassCastException` – an attempt is made to cast an object that does not have an *is-a* relationship with the type specified in the cast operator
- `NullPointerException` – when a `null` reference is used where an object is expected
- ArithmeticException – can arise from a number of different problems in arithmetic
- Throw point – initial point at which the exception occurs, top row of call chain
- InputMismatchException – occurs when Scanner method nextInt receives a string that does not represent a valid integer

Exception Handling

- Exception handling – resolving exceptions that may occur so program can continue or terminate gracefully

- Exception handling enables programmers to create programs that are more robust and fault-tolerant
- Improves clarity
- Enhances modifiability

Divide by Zero with no exception handling

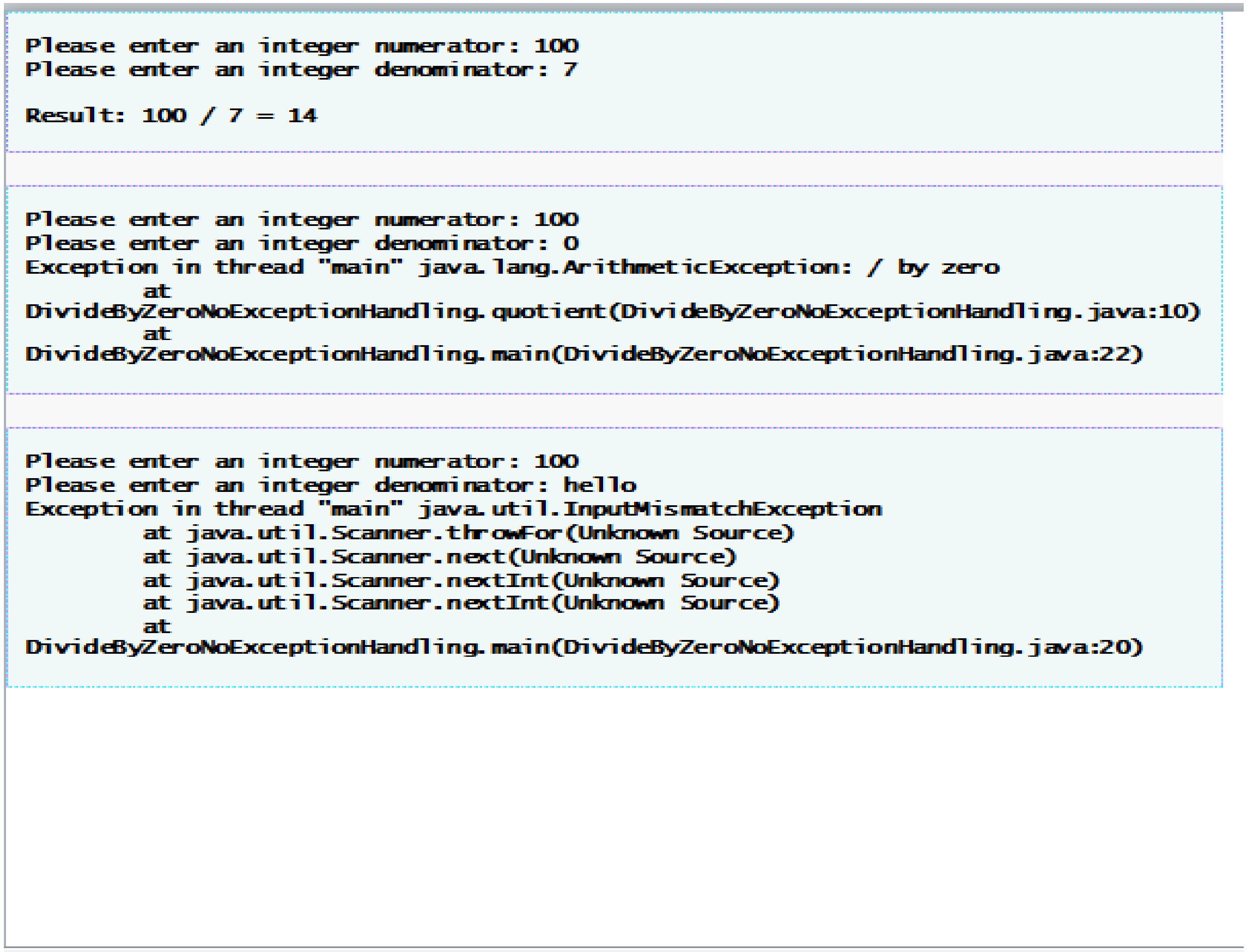
```java
// Fig. 13.1: DivideByZeroNoExceptionHandling.java
// An application that attempts to divide by zero.
import java.util.Scanner;

public class DivideByZeroNoExceptionHandling
{
   // demonstrates throwing an exceptio
   public static int quotient( int numer
   {
      return numerator / denominator; // possible division by zero
   } // end method quotient

   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input

      System.out.print( "Please enter an integer numerator: " );
      int numerator = scanner.nextInt();
      System.out.print( "Please enter an integer
      int denominator = scanner.nextInt();

      int result = quotient( numerator, denominator );
      System.out.printf(
         "\nResult: %d / %d = %d\n", numerator, denominator, result );
   } // end main
} // end class DivideByZeroNoExceptionHandling
```

Attempt to divide; denominator may be zero

Read input; exception occurs if input is not a valid integer

```
Please enter an integer numerator: 100
Please enter an integer denominator: 7
```

Divide by Zero with no exception handling



```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at
DivideByZeroNoExceptionHandling.quotient(DivideByZeroNoExceptionHandling.java:10)
        at
DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:22)
```

```
Please enter an integer numerator: 100
Please enter an integer denominator: hello
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Unknown Source)
        at java.util.Scanner.next(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at java.util.Scanner.nextInt(Unknown Source)
        at
DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:20)
```

Example: Default Exception Handling

Displays this error message

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero at
divByZero.main(DivByZero.java:10)
```

Default exception handler

- Provided by Java runtime
- Prints out exception description
- Prints the stack trace
- Hierarchy of methods where the exception occurred

– Causes the program to terminate

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>
}
```

Enclosing code in a try block

- `try` block – encloses code that might throw an exception and the code that should not execute if an exception occurs

- Consists of keyword `try` followed by a block of code enclosed in curly braces

Catching exceptions

- `catch` block – catches (i.e., receives) and handles an exception:
  - Begins with keyword `catch`
  - Exception parameter in parentheses – exception parameter identifies the exception type and enables `catch` block to interact with caught exception object
  - Block of code in curly braces that executes when exception of proper type occurs
- Matching `catch` block – the type of the exception parameter matches the thrown exception type exactly or is a superclass of it
- Uncaught exception – an exception that occurs for which there are no matching `catch` blocks
  - Cause program to terminate if program has only one thread; Otherwise only current thread is terminated and there may be adverse effects to the rest of the program

Termination Model of Exception Handling

- When an exception occurs:
  - `try` block terminates immediately
  - Program control transfers to first matching `catch` block
- After exception is handled:
  - Termination model of exception handling – program control does not return to the throw point because the `try` block has expired; Flow of control proceeds to the first statement after the last `catch` block
  - Resumption model of exception handling – program control resumes just after throw point
- `try` statement – consists of `try` block and corresponding `catch` and/or `finally` blocks

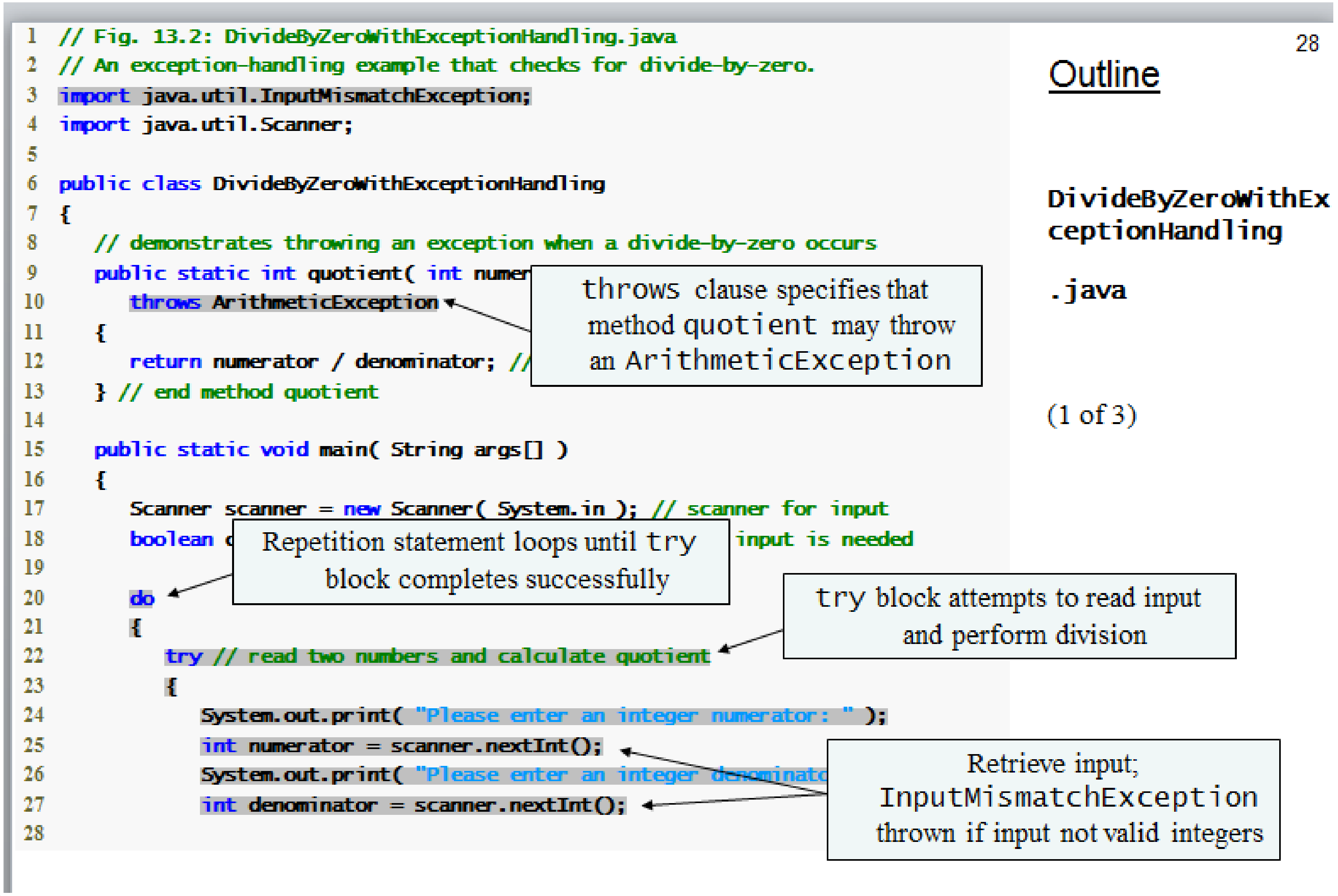
Using the throws clause

- `throws` clause – specifies the exceptions a method may throws
  - Appears after method’s parameter list and before the method’s body
  - Contains a comma-separated list of exceptions
  - Exceptions can be thrown by statements in method’s body of by methods called in method’s body
  - Exceptions can be of types listed in `throws` clause or subclasses

Catching Exceptions: The *try-catch* Statements

```java
class DivByZero {
    public static void main(String args[]) {
        try {
            System.out.println(3/0);
            System.out.println("Please print me.");
        } catch (ArithmeticException exc) {
            //Division by zero is an ArithmeticException
            System.out.println(exc);
        }
        System.out.println("After exception.");
    }
}
```

Handling ArithmeticExceptions and InputMismatchExceptions



```
// Fig. 13.2: DivideByZeroWithExceptionHandling.java
// An exception-handling example that checks for divide-by-zero.
import java.util.InputMismatchException;
import java.util.Scanner;

public class DivideByZeroWithExceptionHandling
{
   // demonstrates throwing an exception when a divide-by-zero occurs
   public static int quotient( int numer
      throws ArithmeticException
   {
      return numerator / denominator; //
   } // end method quotient

   public static void main( String args[] )
   {
      Scanner scanner = new Scanner( System.in ); // scanner for input
      boolean c                                    input is needed

      do
      {
         try // read two numbers and calculate quotient
         {
            System.out.print( "Please enter an integer numerator: " );
            int numerator = scanner.nextInt();
            System.out.print( "Please enter an integer denominato
            int denominator = scanner.nextInt();

```

`throws` clause specifies that method `quotient` may throw an `ArithmeticException`

Repetition statement loops until `try` block completes successfully

`try` block attempts to read input and perform division

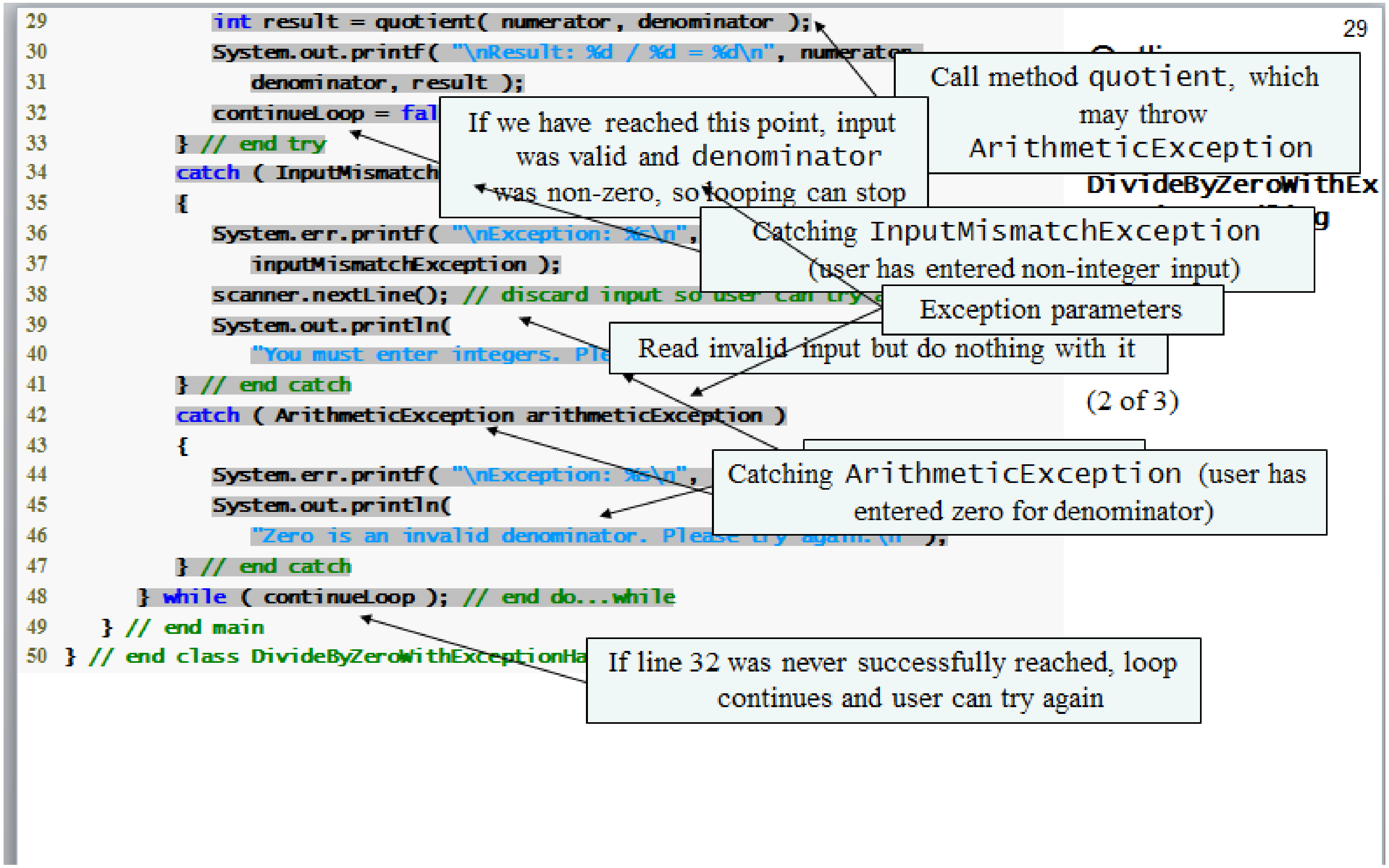
Retrieve input; `InputMismatchException` thrown if input not valid integers

Outline

`DivideByZeroWithExceptionHandling.java`

(1 of 3)

Handling ArithmeticExceptions and InputMismatchExceptions



29

```
29              int result = quotient( numerator, denominator );
30              System.out.printf( "\nResult: %d / %d = %d\n", numerator,
31                 denominator, result );
32              continueLoop = fal
33           } // end try
34           catch ( InputMismatch
35           {
36              System.err.printf( "\nException: %s\n",
37                 inputMismatchException );
38              scanner.nextLine(); // discard input so user can try a
39              System.out.println(
40                 "You must enter integers. Ple
41           } // end catch
42           catch ( ArithmeticException arithmeticException )
43           {
44              System.err.printf( "\nException: %s\n",
45              System.out.println(
46                 "Zero is an invalid denominator. Please try again.\n" );
47           } // end catch
48        } while ( continueLoop ); // end do...while
49     } // end main
50  } // end class DivideByZeroWithExceptionHa
```

Call method `quotient`, which may throw `ArithmeticException`

If we have reached this point, input was valid and `denominator` was non-zero, so looping can stop

DivideByZeroWithEx

Catching `InputMismatchException` (user has entered non-integer input)

Exception parameters

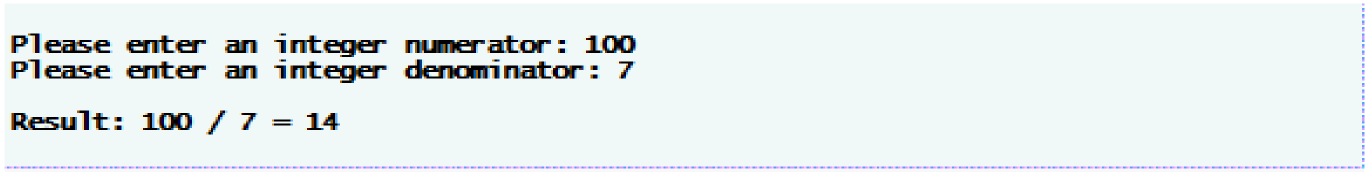
Read invalid input but do nothing with it

(2 of 3)

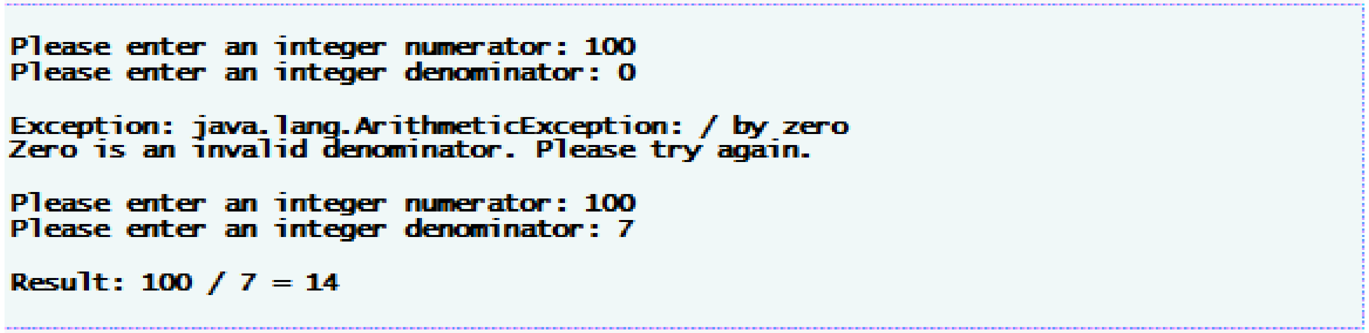
Catching `ArithmeticException` (user has entered zero for denominator)

If line 32 was never successfully reached, loop continues and user can try again

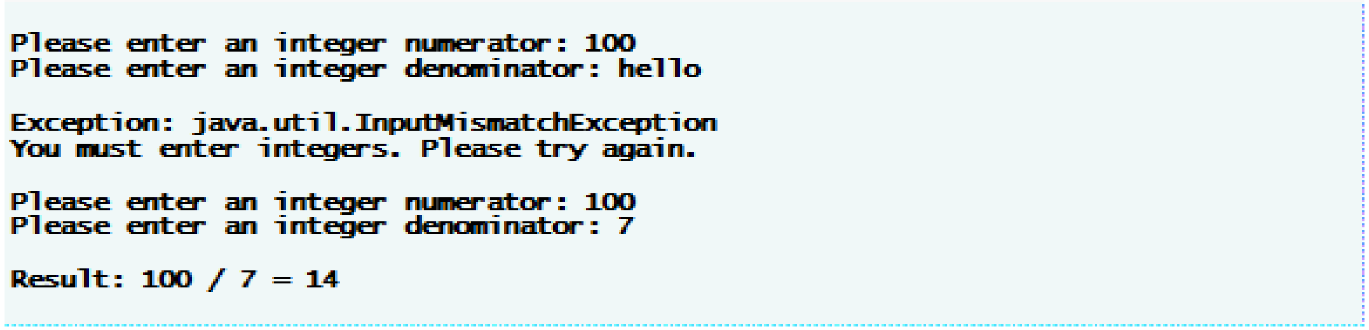
Handling ArithmeticExceptions and InputMismatchExceptions



```
Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```



```
Please enter an integer numerator: 100
Please enter an integer denominator: 0

Exception: java.lang.ArithmeticException: / by zero
Zero is an invalid denominator. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```



```
Please enter an integer numerator: 100
Please enter an integer denominator: hello

Exception: java.util.InputMismatchException
You must enter integers. Please try again.

Please enter an integer numerator: 100
Please enter an integer denominator: 7

Result: 100 / 7 = 14
```

Catching Exceptions: Multiple catch

```
class MultipleCatch {
    public static void main(String args[]) {
    try {
    int den = Integer.parseInt(args[0]);
        System.out.println(3/den);
    }
    catch (ArithmeticException exc) {
        System.out.println("Divisor was 0.");
    } catch (ArrayIndexOutOfBoundsException exc2) {
    System.out.println("Missing argument.");
    }
    System.out.println("After exception.");
}   }
```

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!");
}
}
}
```

13.6 Java Exception Hierarchy

- All exceptions inherit either directly or indirectly from class `Exception`
- Exception classes form an inheritance hierarchy that can be extended
- Class `Throwable`, superclass of `Exception`
  - Only `Throwable` objects can be used with the exception-handling mechanism
  - Has two subclasses: `Exception` and `Error`
    - Class `Exception` and its subclasses represent exception situations that can occur in a Java program and that can be caught by the application
    - Class `Error` and its subclasses represent abnormal situations that could happen in the JVM – it is usually not possible for a program to recover from `Error`s

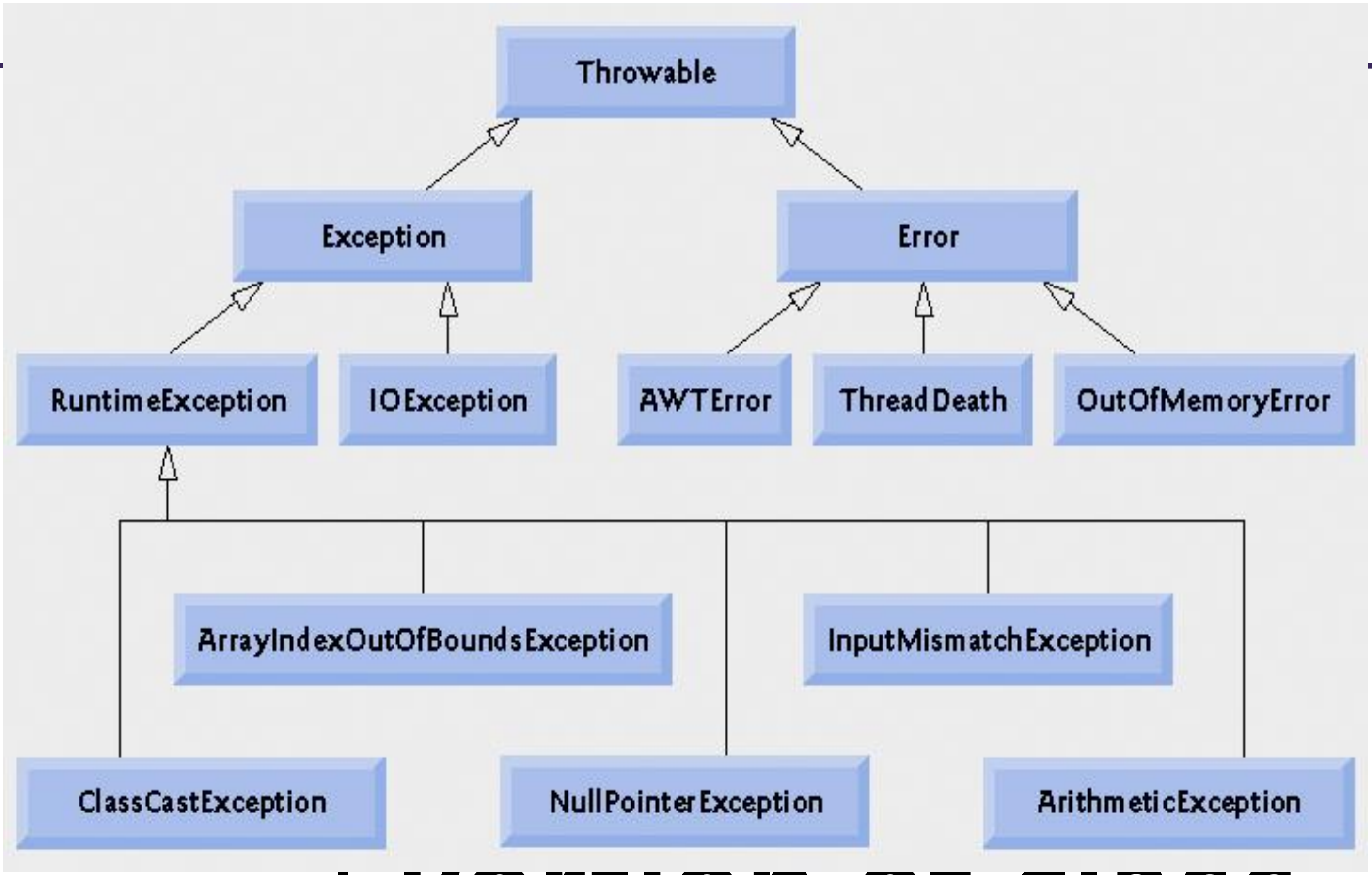


Fig. 13.3 | Portion of class Throwable's inheritance hierarchy.

Java Exception Hierarchy

- Two categories of exceptions: checked and unchecked
- Checked exceptions
  - Exceptions that inherit from class `Exception` but not from `RuntimeException`
  - Compiler enforces a catch-or-declare requirement
  - Compiler checks each method call and method declaration to determine whether the method `throws` checked exceptions. If so, the compiler ensures that the checked exception is caught or is declared in a `throws` clause. If not caught or declared, compiler error occurs.
- Unchecked exceptions
  - Inherit from class `RuntimeException` or class `Error`
  - Compiler does not check code to see if exception is caught or declared
  - If an unchecked exception occurs and is not caught, the program terminates or runs with unexpected results
  - Can typically be prevented by proper coding

`finally` block

- Programs that obtain certain resources must return them explicitly to avoid resource leaks
- `finally` block
  - Consists of `finally` keyword followed by a block of code enclosed in curly braces
  - Optional in a `try` statement
  - If present, is placed after the last `catch` block
  - Executes whether or not an exception is thrown in the corresponding `try` block or any of its corresponding `catch` blocks
  - Will not execute if the application exits early from a `try` block via method `System.exit`
  - Typically contains resource-release code

Outline

```
try
{
   statements
   resource-acquisition statements
} // end try
catch ( AKindOfException  exception1 )
{
   exception-handling statements
} // end catch
.
.
.
catch ( AnotherKindOfException  exception2 )
{
   exception-handling statements
} // end catch
finally
{
   statements
   resource-release statements
} // end finally
```

Position of the finally block after the last catch block in a try statement.

`finally` block

- If no exception occurs, `catch` blocks are skipped and control proceeds to `finally` block.
- After the `finally` block executes control proceeds to first statement after the `finally` block.
- If exception occurs in the `try` block, program skips rest of the `try` block. First matching the `catch` block executes and control proceeds to the `finally` block.
- If exception occurs and there are no matching `catch` blocks, control proceeds to the `finally` block. After the `finally` block executes, the program passes the exception to the next outer the `try` block.
- If `catch` block throws an exception, the `finally` block still executes.

`finally` block

- Standard streams
  - `System.out` – standard output stream
  - `System.err` – standard error stream
- `System.err` can be used to separate error output from regular output
- `System.err.println` and `System.out.println` display data to the command prompt by default
- E.g

```
finally {
    System.out.println("try-block entered.");
}
```