Data Structures

02 – Exception Handling

David Drohan

JAVA: An Introduction to Problem Solving & Programming, 6th Ed. By Walter Savitch
ISBN 0132162709 © 2012 Pearson Education, Inc., Upper Saddle River, NJ. All Rights Reserved

Objectives

- ❑ Describe the notion of exception handling
- ❑ React correctly when certain exceptions occur
- ❑ Use Java's exception-handling facilities effectively in classes and programs
- ❑ Create & use custom exceptions

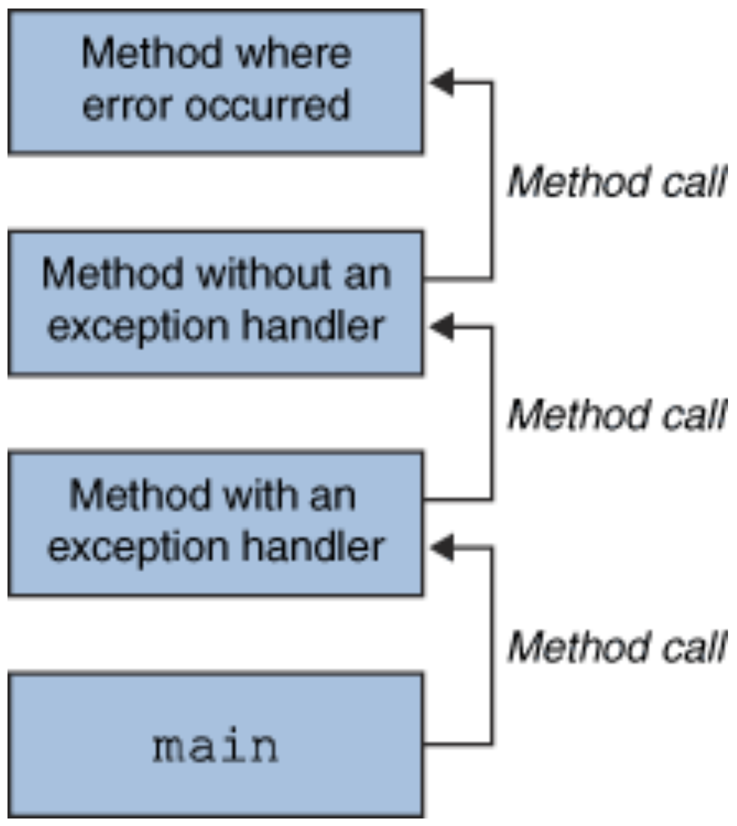
What is an Exception?

- The term *exception* is shorthand for the phrase "exceptional event."
- **Definition:** An *exception* is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- When an error occurs within a method, the method creates an **object** and hands it off to the runtime system. The object, called an *exception object*, contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called *throwing an exception*.

What is an Exception?

- After a method throws an exception, the runtime system attempts to find something to handle it. The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred. The list of methods is known as the ***call stack*** (see the next figure).

The Call Stack


Method where error occurred
Method call
Method without an exception handler
Method call
Method with an exception handler
Method call
main

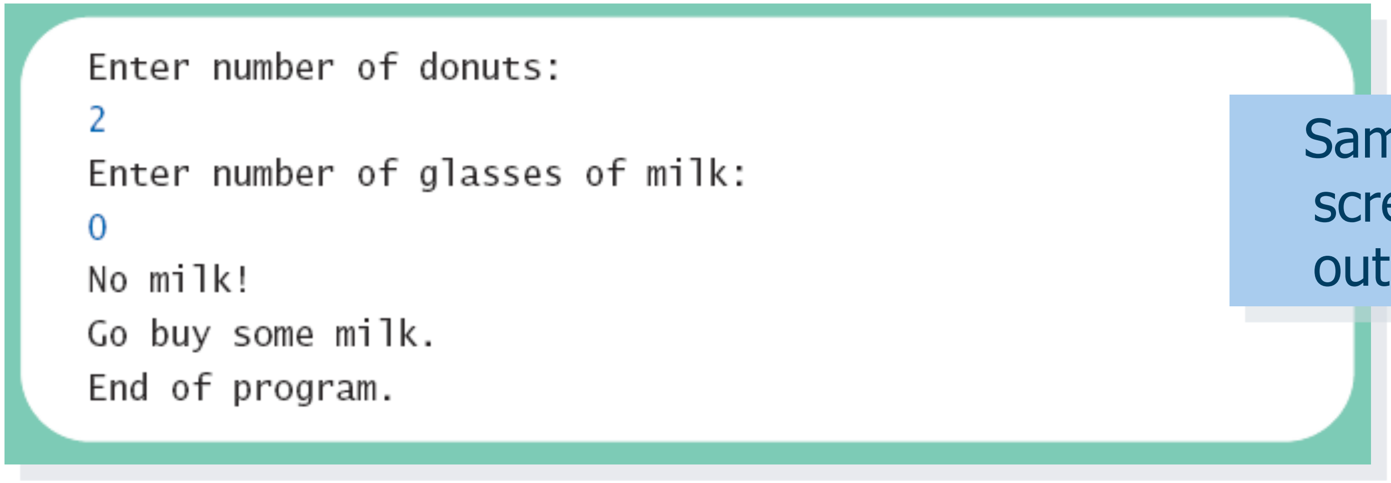
Basic Exception Handling

Exceptions in Java

- An exception is an object
  - Signals the occurrence of unusual event during program execution
- Throwing an exception
  - Creating the exception object
- Handling the exception
  - Code that detects and deals with the exception

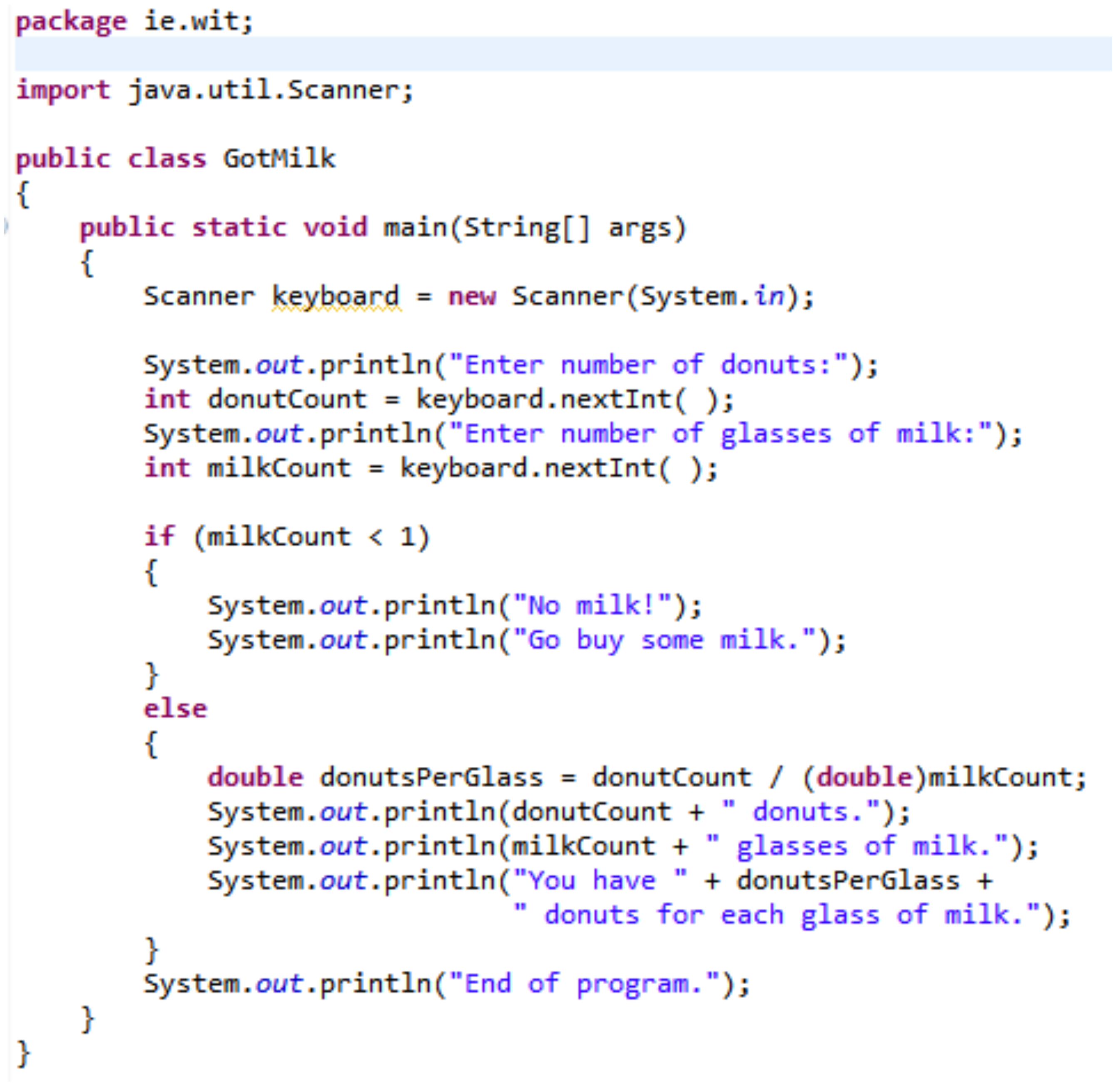
Exceptions in Java

- Consider a program to assure us of a sufficient supply of milk
- Possible solution, ***`class GotMilk`*** (next slide)



```
Enter number of donuts:
2
Enter number of glasses of milk:
0
No milk!
Go buy some milk.
End of program.
```

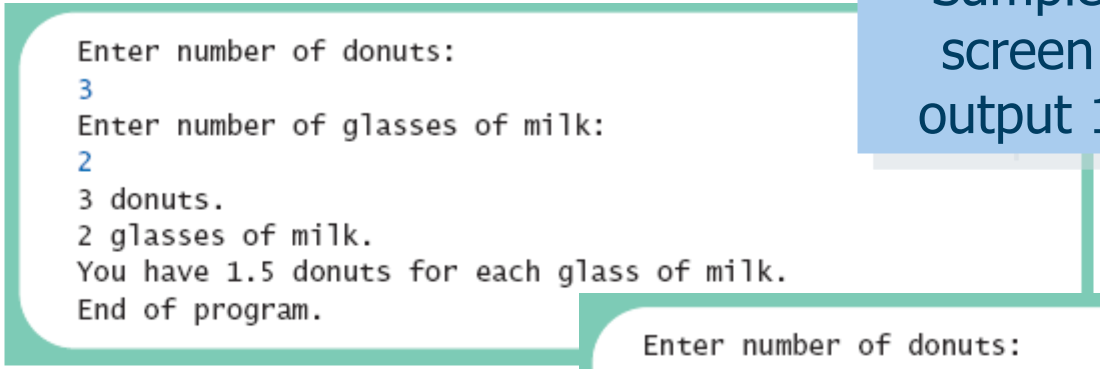
Sample screen output



```java
package ie.wit;

import java.util.Scanner;

public class GotMilk
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Enter number of donuts:");
        int donutCount = keyboard.nextInt( );
        System.out.println("Enter number of glasses of milk:");
        int milkCount = keyboard.nextInt( );

        if (milkCount < 1)
        {
            System.out.println("No milk!");
            System.out.println("Go buy some milk.");
        }
        else
        {
            double donutsPerGlass = donutCount / (double)milkCount;
            System.out.println(donutCount + " donuts.");
            System.out.println(milkCount + " glasses of milk.");
            System.out.println("You have " + donutsPerGlass +
                               " donuts for each glass of milk.");
        }
        System.out.println("End of program.");
    }
}
```

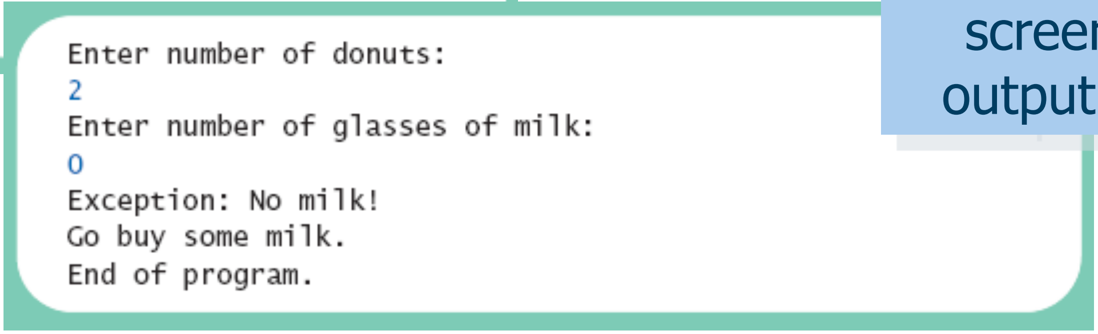
Exceptions in Java

- Now we revise the program to use exception-handling
- View new version, ***`class ExceptionDemo`***



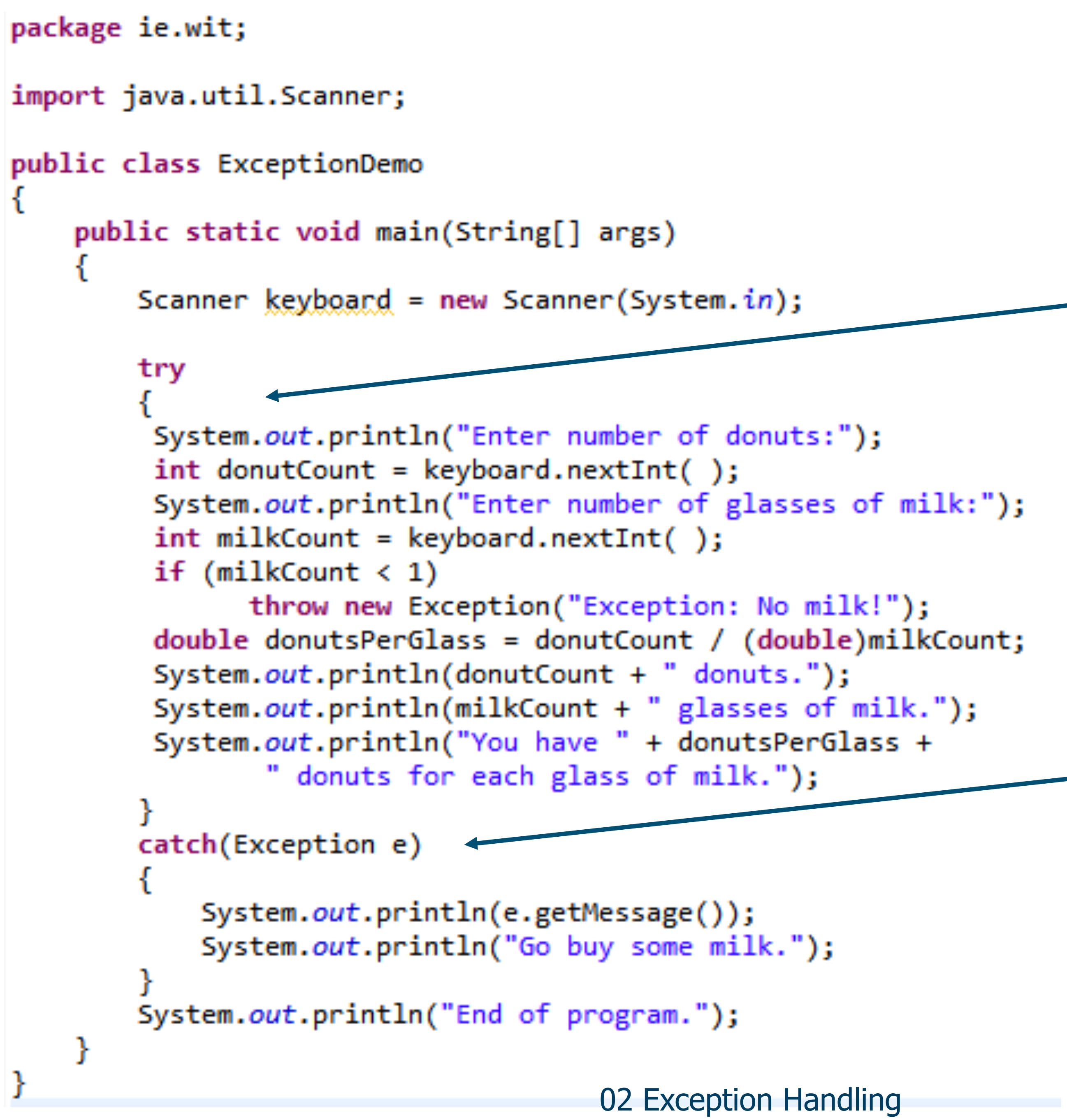
```
Enter number of donuts:
3
Enter number of glasses of milk:
2
3 donuts.
2 glasses of milk.
You have 1.5 donuts for each glass of milk.
End of program.
```

Sample screen output 1



```
Enter number of donuts:
2
Enter number of glasses of milk:
0
Exception: No milk!
Go buy some milk.
End of program.
```

Sample screen output 2



```java
package ie.wit;

import java.util.Scanner;

public class ExceptionDemo
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);

        try
        {
         System.out.println("Enter number of donuts:");
         int donutCount = keyboard.nextInt( );
         System.out.println("Enter number of glasses of milk:");
         int milkCount = keyboard.nextInt( );
         if (milkCount < 1)
               throw new Exception("Exception: No milk!");
         double donutsPerGlass = donutCount / (double)milkCount;
         System.out.println(donutCount + " donuts.");
         System.out.println(milkCount + " glasses of milk.");
         System.out.println("You have " + donutsPerGlass +
                " donuts for each glass of milk.");
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
            System.out.println("Go buy some milk.");
        }
        System.out.println("End of program.");
    }
}
```

'`try`' block

'`catch`' block

Exceptions in Java

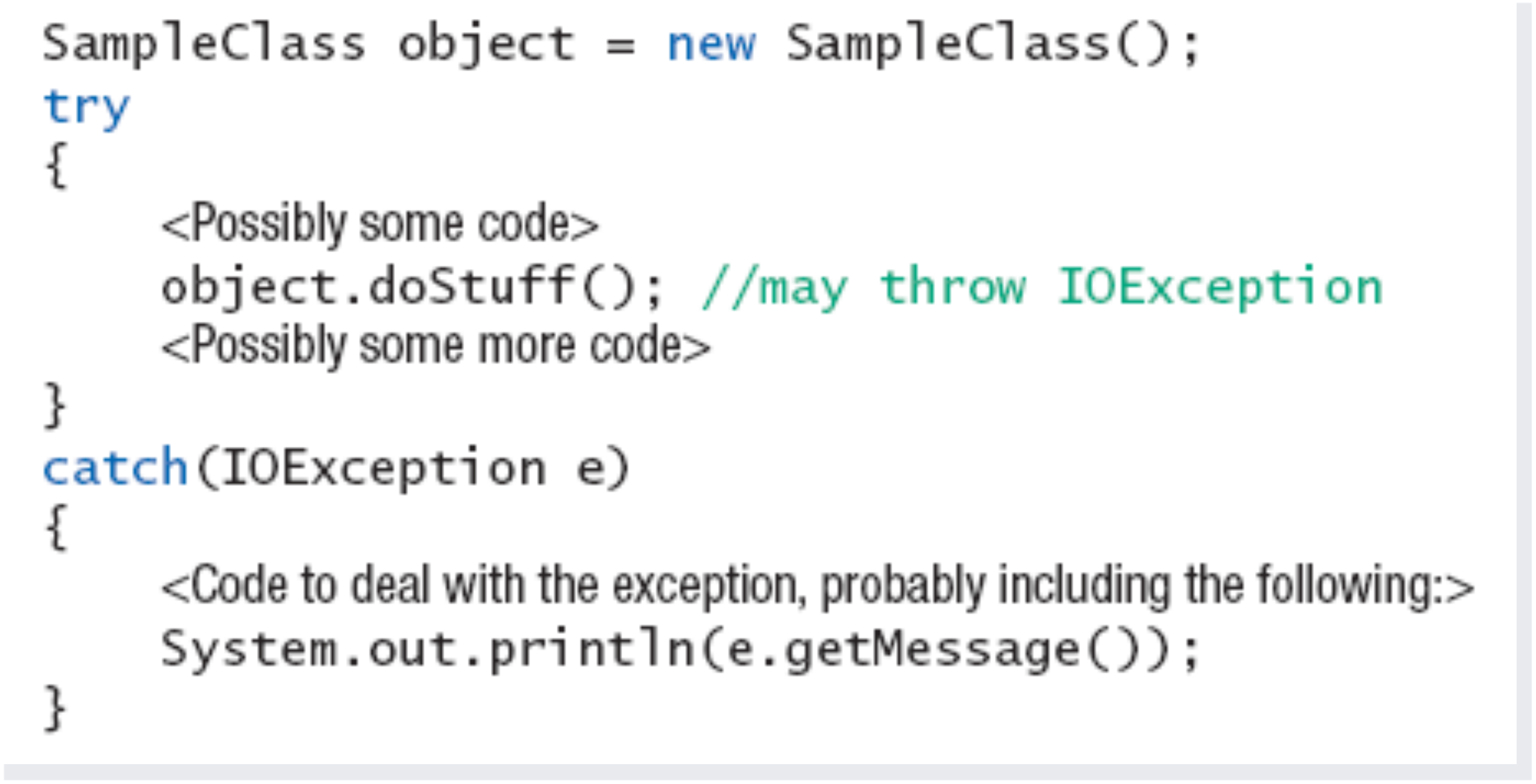
- Note `try` block
  - Contains code where something could possibly go wrong
  - If it does go wrong, we ***throw an exception***
- Note `catch` block
  - When exception thrown, `catch` block begins execution
  - Similar to method with parameter
  - Parameter ***is*** the thrown object

Predefined Exception Classes

- Java has predefined exception classes within Java Class Library
  - You can place method invocation in `try` block
  - You follow with `catch` block for this type of exception
- Example classes
  - `NullPointerException`
  - `ArrayIndexOutOfBoundsException`
  - `ClassNotFoundException`
  - `IOException`
  - `NoSuchMethodException`

Predefined Exception Classes

- Example code



```
SampleClass object = new SampleClass();
try
{
      <Possibly some code>
      object.doStuff(); //may throw IOException
      <Possibly some more code>
}
catch(IOException e)
{
      <Code to deal with the exception, probably including the following:>
      System.out.println(e.getMessage());
}
```

More About Exception Classes

Outline

- Declaring Exceptions (Passing the Buck)
- Kinds of Exceptions
- Errors
- Multiple Throws and Catches
- The `finally` Block
- Rethrowing an Exception
- Case Study : Custom Exceptions

Declaring Exceptions

- Consider a method where its code throws an exception
  - May want to handle immediately
  - May want to delay until something else is done
- Method that does not catch an exception
  - Notify programmers (users) of your method with `throws` clause
  - Programmer then given responsibility to handle exception

Declaring Exceptions

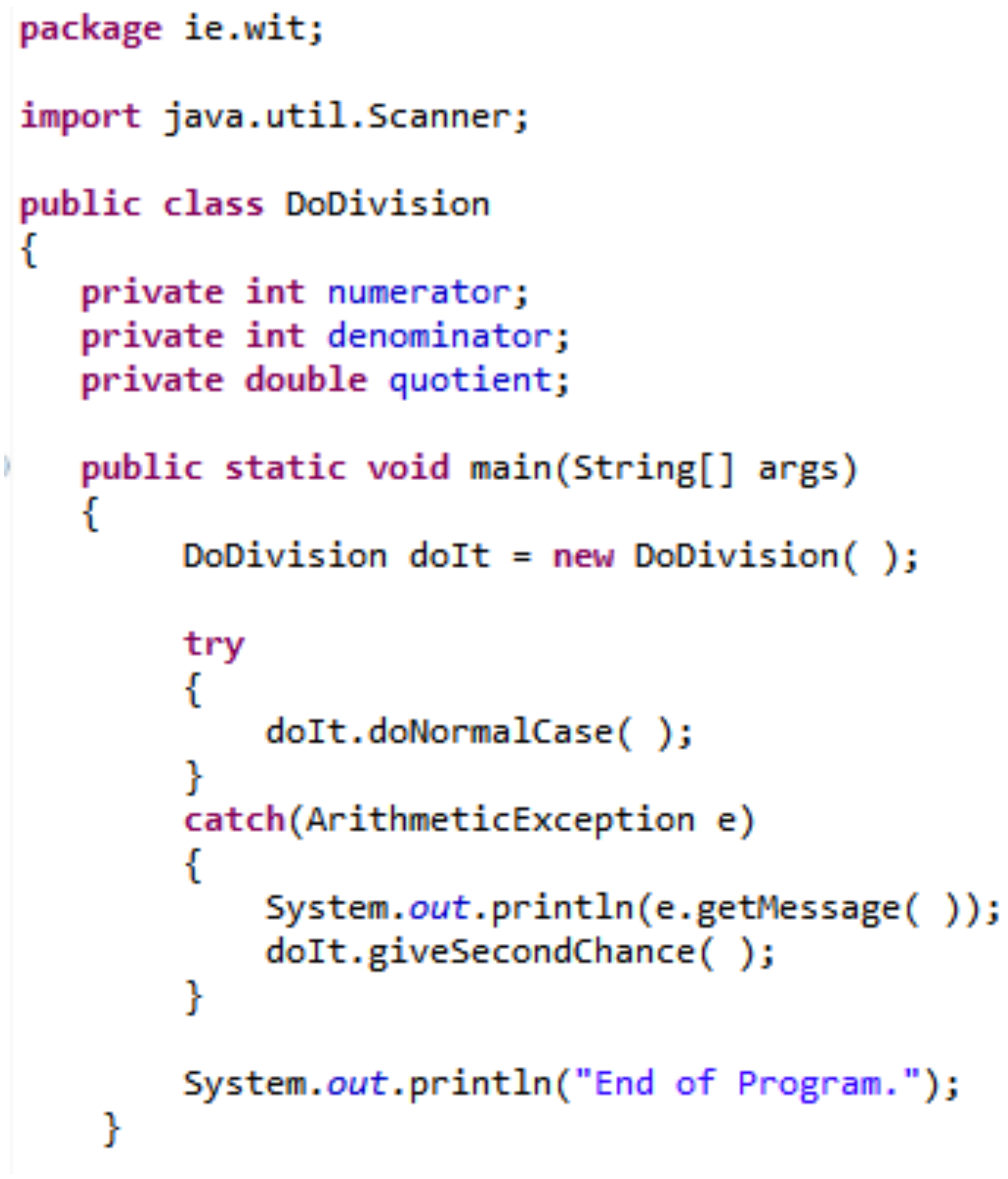
- Note syntax for throws clause

```
public Type Method_Name(Parameter_List) throws List_Of_Exceptions
Body_Of_Method
```

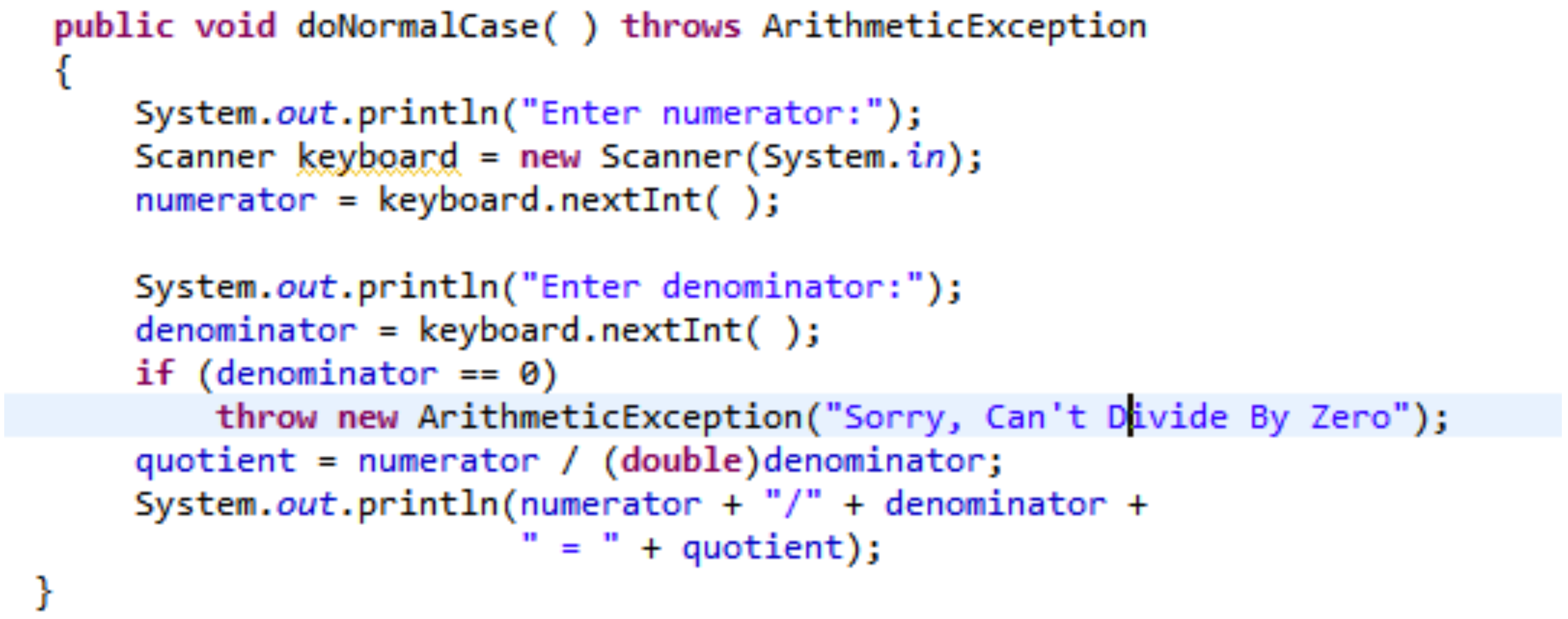
- Note distinction
  - Keyword `throw` used to throw exception
  - Keyword `throws` used in method header to declare an exception

Declaring Exceptions

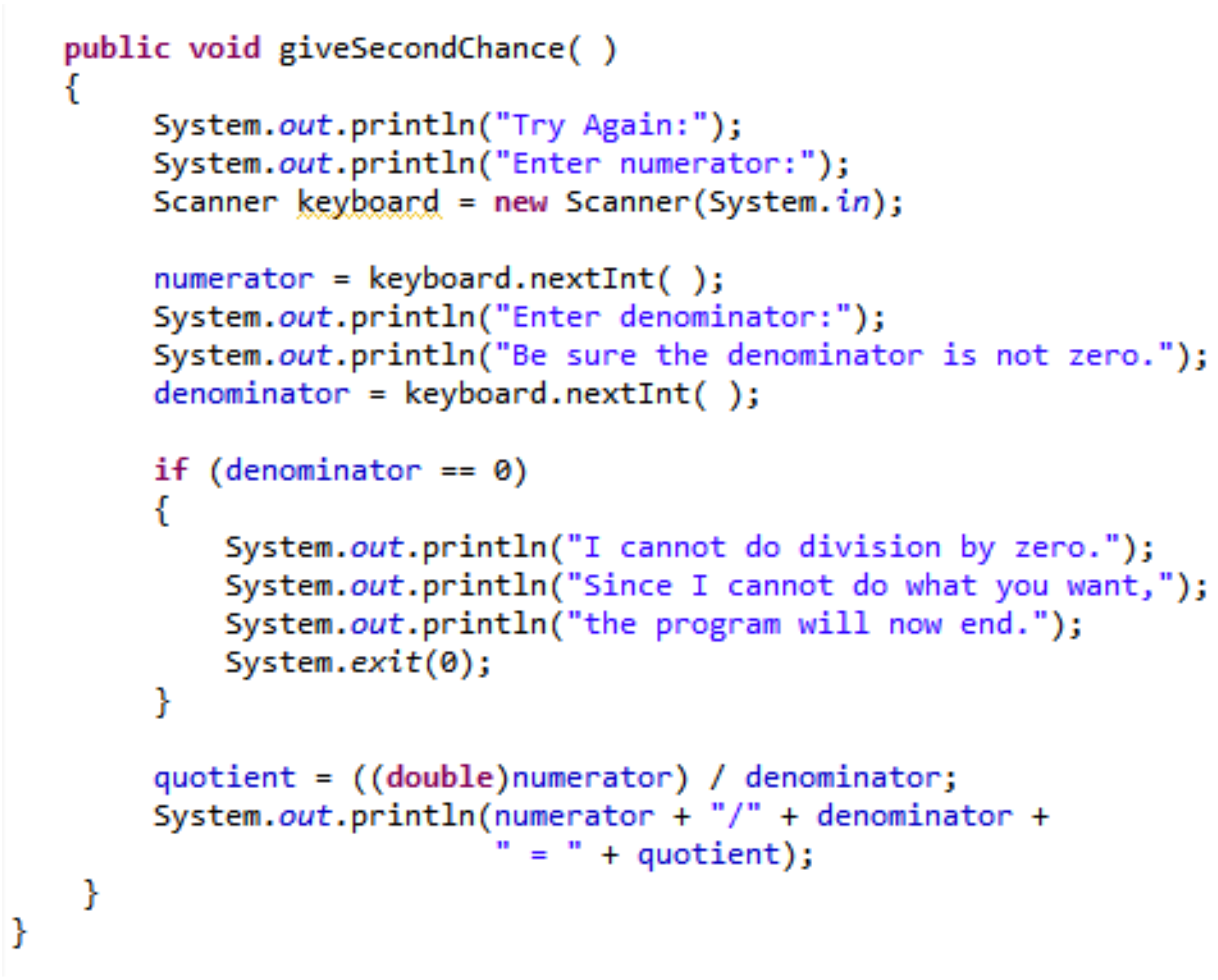
- If a method throws an exception and the exception is not caught inside the method
  - Method ends immediately after exception thrown
- A throws clause in overriding method
  - Can declare fewer exceptions than declared
  - But not more
- View program example, ***`class DoDivision`***



```java
package ie.wit;

import java.util.Scanner;

public class DoDivision
{
   private int numerator;
   private int denominator;
   private double quotient;

   public static void main(String[] args)
   {
        DoDivision doIt = new DoDivision( );

        try
        {
            doIt.doNormalCase( );
        }
        catch(ArithmeticException e)
        {
            System.out.println(e.getMessage( ));
            doIt.giveSecondChance( );
        }

        System.out.println("End of Program.");
    }
```



```java
public void doNormalCase( ) throws ArithmeticException
{
    System.out.println("Enter numerator:");
    Scanner keyboard = new Scanner(System.in);
    numerator = keyboard.nextInt( );

    System.out.println("Enter denominator:");
    denominator = keyboard.nextInt( );
    if (denominator == 0)
        throw new ArithmeticException("Sorry, Can't Divide By Zero");
    quotient = numerator / (double)denominator;
    System.out.println(numerator + "/" + denominator +
                       " = " + quotient);
}
```



```
    public void giveSecondChance( )
    {
        System.out.println("Try Again:");
        System.out.println("Enter numerator:");
        Scanner keyboard = new Scanner(System.in);

        numerator = keyboard.nextInt( );
        System.out.println("Enter denominator:");
        System.out.println("Be sure the denominator is not zero.");
        denominator = keyboard.nextInt( );

        if (denominator == 0)
        {
            System.out.println("I cannot do division by zero.");
            System.out.println("Since I cannot do what you want,");
            System.out.println("the program will now end.");
            System.exit(0);
        }

        quotient = ((double)numerator) / denominator;
        System.out.println(numerator + "/" + denominator +
                               " = " + quotient);
     }
}
```

Kinds of Exceptions

- In most cases, exception is caught either….
  - In a `catch` block … or
  - Be declared in a `throws` clause
- But Java has exceptions you do not need to account for
- Categories of exceptions
  - Checked exceptions
  - Unchecked exceptions

Kinds of Exceptions

- *Checked* exception
  - **Must** be caught in a `catch` block
  - Or declared in a `throws` clause
- *Unchecked* exception
  - Also called *run-time*
  - Need not be caught in `catch` block or declared in `throws`
  - However, exceptions that highlight coding problems should be fixed (obviously…)

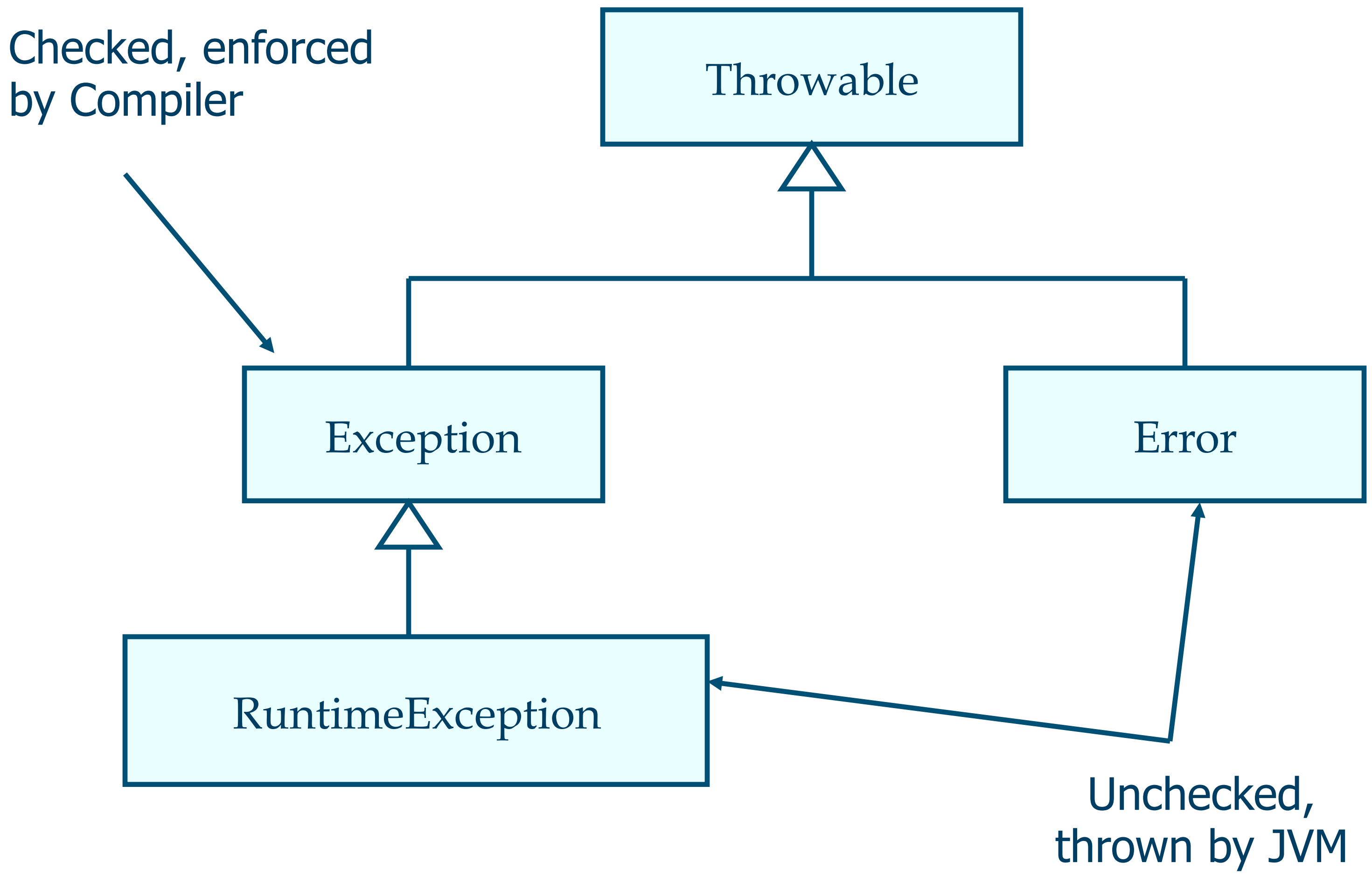
Checked Exceptions

- Exceptions checked by compiler
  - Need to be handled in the code as Compiler gives a compile error if they are not handled
  - Exceptions are thrown by methods that are used in the code
    - That’s how Compiler recognizes them
- What happens when exceptions are caught depends on the exception handling strategy
  - Exception handling code defines how exceptions are handled
- Examples - The JDBC API & File I/O

Unchecked Exceptions

- Exceptions that are thrown by the Java Virtual Machine
  - Cannot be handled at the compilation as compiler does not enforce developers to handle exceptions
  - These exceptions occur at the runtime
  - Hard to catch specific unchecked exception as they are not checked by the compiler
  - Example – Basic I/O, Numbers Vs Text (look at the Scanner class….), Attempt to use array index out of bounds, Division by zero

Exception Hierarchy…



…Exception Hierarchy

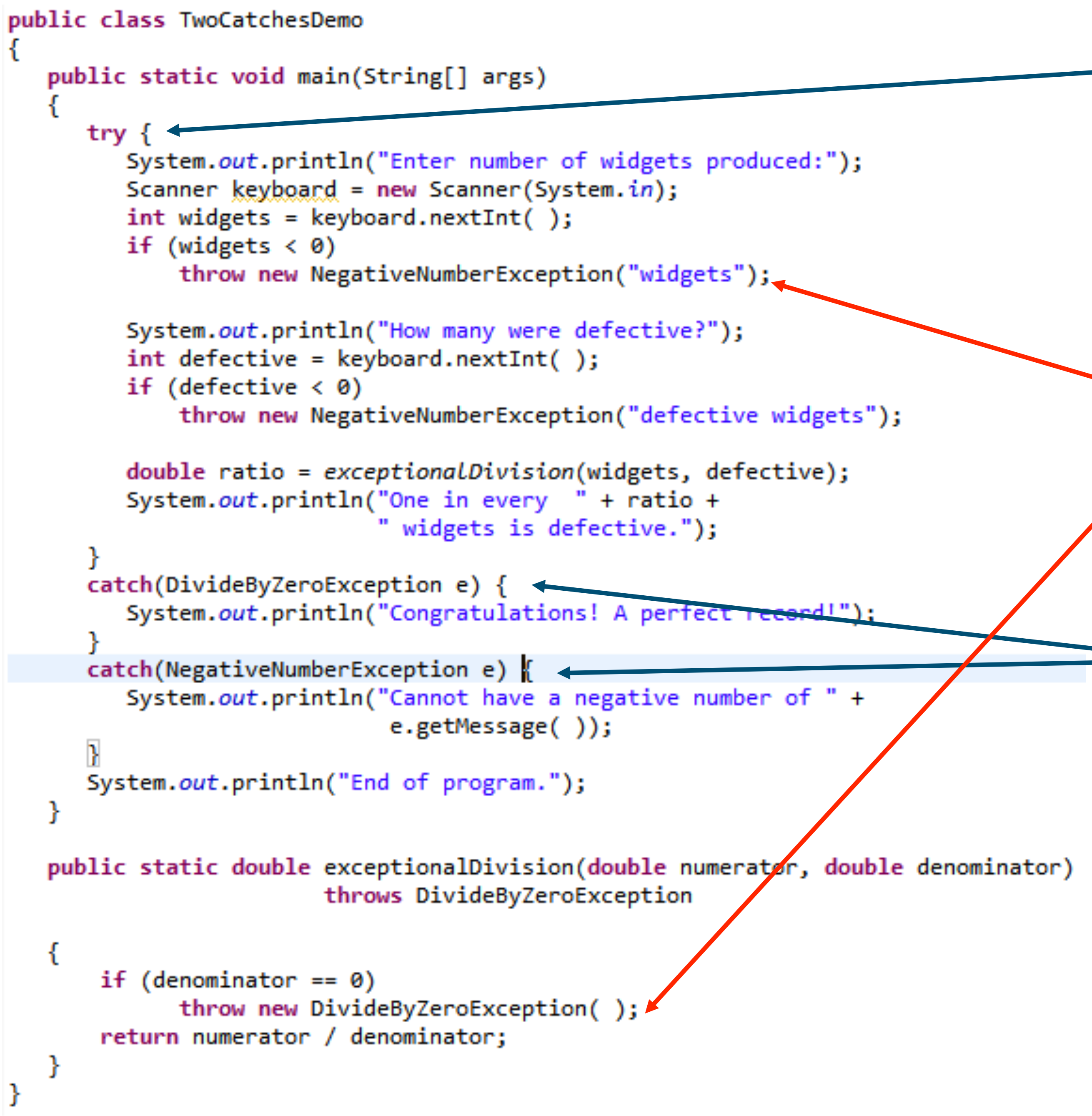
- Throwable – top of the exception hierarchy in Java, all exceptions are of this type
- Error – represents serious problems in program, that usually cannot be recovered from; thrown by the JVM (see next slide)
- Exception – superclass for all exceptions including user-defined exceptions
- RuntimeException – also thrown by JVM and caused by illegal operations

Some Common Java **Errors**

- NoSuchMethodError
  - Application calls method that no longer exist in the class definition
    - Usually happens if class definition changes at runtime
- NoClassDefFoundError
  - JVM tries to load class and class cannot be found
    - Usually happens if classpath is not set, or class somehow gets removed from the classpath
- ClassFormatError
  - JVM tries to load class from file that is incorrect
    - Usually happens if class file is corrupted, or if it isn't class file

Multiple Throws and Catches

- A try block can throw any number of exceptions of different types
- Each catch block can catch exceptions of only **one** type
  - Order of catch blocks matter! – why??
- View example program, ***`class TwoCatchesDemo`***



```java
public class TwoCatchesDemo
{
   public static void main(String[] args)
   {
      try {
         System.out.println("Enter number of widgets produced:");
         Scanner keyboard = new Scanner(System.in);
         int widgets = keyboard.nextInt( );
         if (widgets < 0)
             throw new NegativeNumberException("widgets");

         System.out.println("How many were defective?");
         int defective = keyboard.nextInt( );
         if (defective < 0)
             throw new NegativeNumberException("defective widgets");

         double ratio = exceptionalDivision(widgets, defective);
         System.out.println("One in every  " + ratio +
                              " widgets is defective.");
      }
      catch(DivideByZeroException e) {
         System.out.println("Congratulations! A perfect record!");
      }
      catch(NegativeNumberException e) {
         System.out.println("Cannot have a negative number of " +
                              e.getMessage( ));
      }
      System.out.println("End of program.");
   }

   public static double exceptionalDivision(double numerator, double denominator)
                     throws DivideByZeroException

   {
       if (denominator == 0)
             throw new DivideByZeroException( );
       return numerator / denominator;
   }
}
```

‘try’ block

‘custom’ exceptions

‘catch’ blocks

Multiple Throws and Catches

- Note multiple sample runs

Sample screen output 1

```
Enter number of widgets produced:
1000
How many were defective?
500
One in every 2.0 widgets is defective.
End of program.
```

Sample screen output 2

```
Enter number of widgets produced:
-10
Cannot have a negative number of widgets
End of program.
```

Sample screen output 3

```
Enter number of widgets produced:
1000
How many were defective?
0
Congratulations! A perfect record!
End of program.
```

Multiple Throws and Catches

- Exceptions can deal with invalid (as apposed to incorrect) user input
- Use of the `throw` statement should be reserved for cases where it is unavoidable
- Convention suggests separate methods for throwing and catching of exceptions
- Nested try-catch blocks rarely useful

The `finally` Block

- Possible to add a `finally` block after sequence of `catch` blocks
- Code in `finally` block executed
  - Whether or not execution thrown
  - Whether or not required `catch` exists

The `finally` block

- Executes always at the end after the last `catch` block (if one exists)
  - Commonly used for cleaning up resources (closing files, streams, etc.)

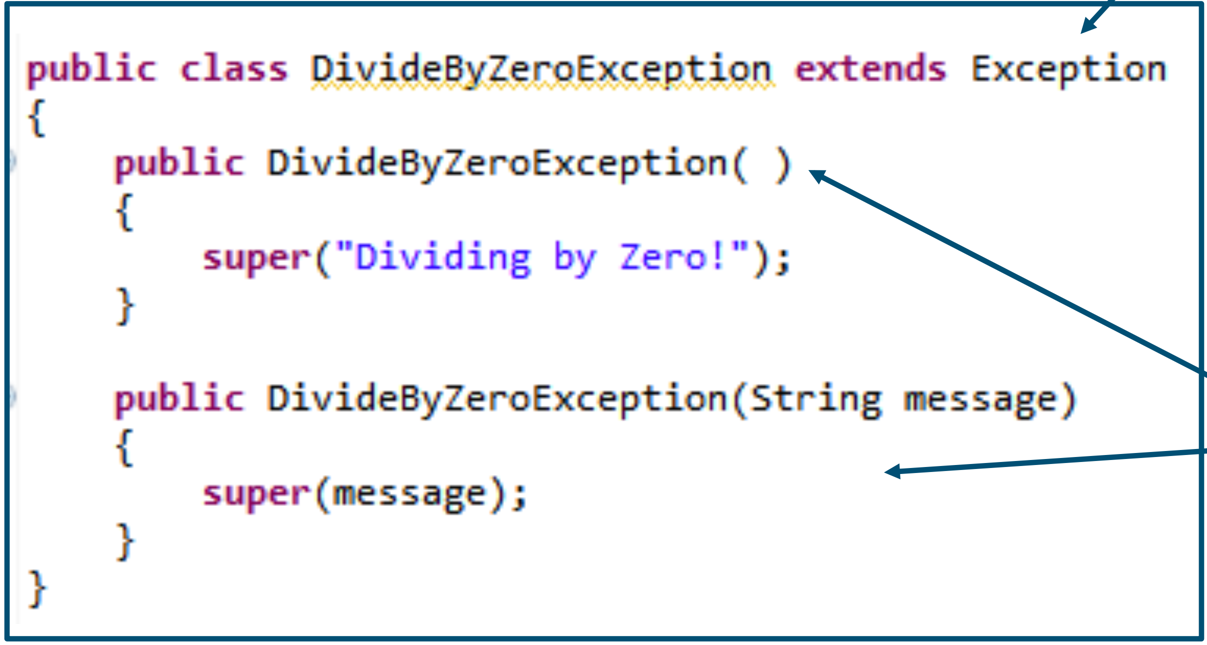
```
public void myMethod()
{
  try{
    //code that throws exception e1
    //code that throws exception e2
  }
  catch (MyException e1){
    //code that handles exception e1
  }
  catch (Exception e2){
    //code that handles exception e2
  }
  finally{
    //clean up code, close resources
  }
}
```

Defining Your Own Exception Classes

Defining Your Own Exception Classes

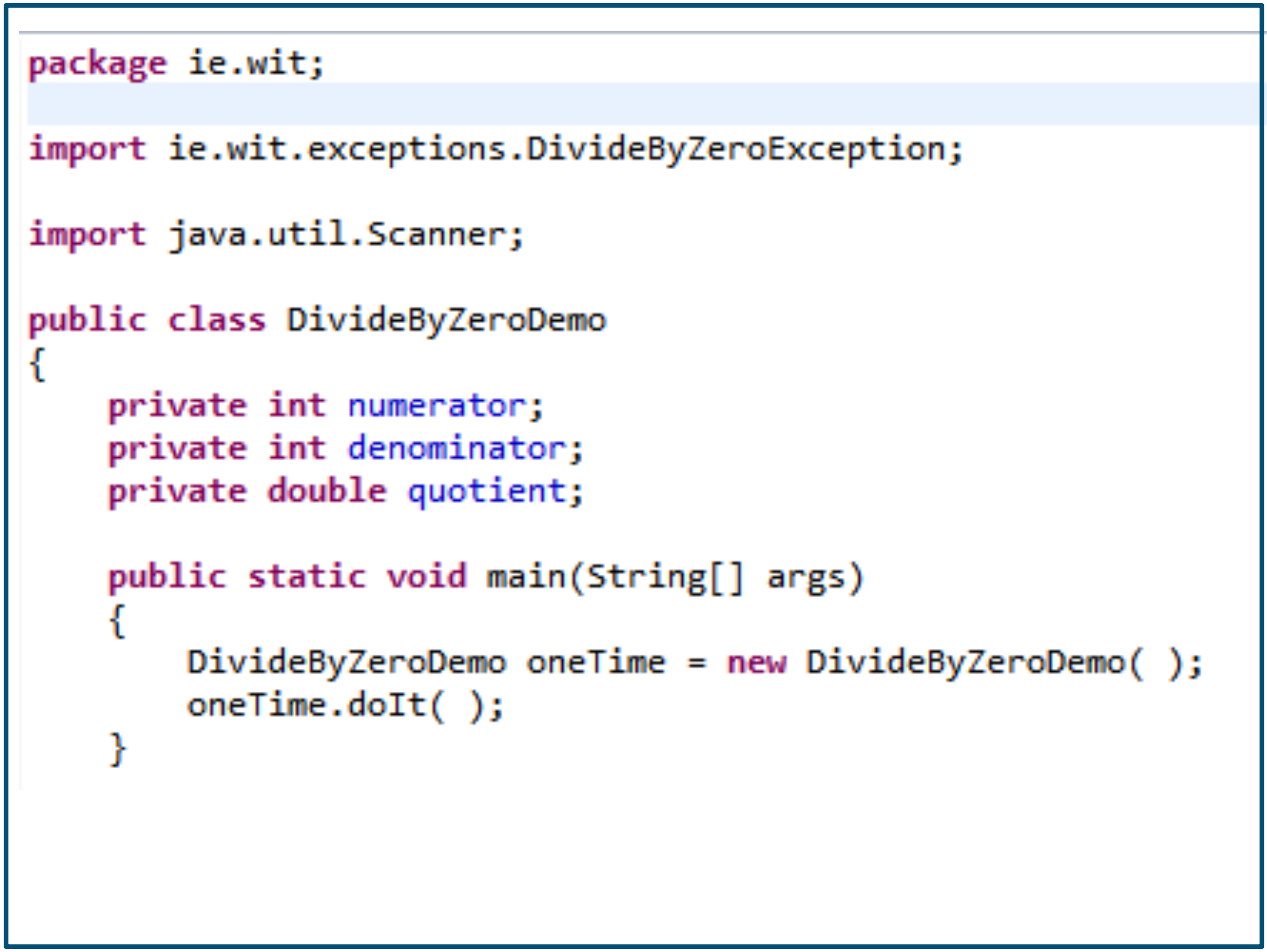
- **Must** be derived class of some predefined exception class
  - Convention suggests use classes derived from class `Exception`
- View sample class
  `class DivideByZeroException`
  and
- View demo program
  `class DivideByZeroDemo`

Extend from Exception



```
public class DivideByZeroException extends Exception
{
    public DivideByZeroException( )
    {
        super("Dividing by Zero!");
    }

    public DivideByZeroException(String message)
    {
        super(message);
    }
}
```

Two constructors



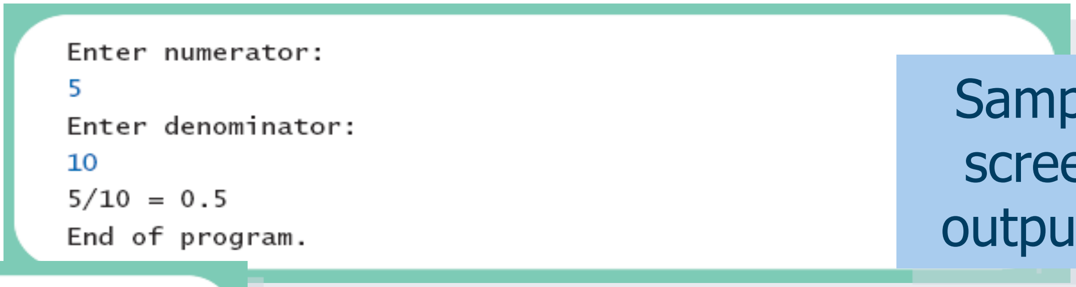
```
package ie.wit;

import ie.wit.exceptions.DivideByZeroException;

import java.util.Scanner;

public class DivideByZeroDemo
{
    private int numerator;
    private int denominator;
    private double quotient;

    public static void main(String[] args)
    {
        DivideByZeroDemo oneTime = new DivideByZeroDemo( );
        oneTime.doIt( );
    }
```

```java
public void doIt( )
{
    try
    {
        System.out.println("Enter numerator:");
        Scanner keyboard = new Scanner(System.in);
        numerator = keyboard.nextInt( );
        System.out.println("Enter denominator:");
        denominator = keyboard.nextInt( );

        if (denominator == 0)
            throw new DivideByZeroException( );

        quotient = numerator / (double)denominator;
        System.out.println(numerator + "/" + denominator +
                                 " = " + quotient);
    }
    catch(DivideByZeroException e)
    {
        System.out.println(e.getMessage( ));
        giveSecondChance( );
    }

    System.out.println("End of program.");
}
```

```java
public void giveSecondChance( )
{
    System.out.println("Try again:");
    System.out.println("Enter numerator:");
    Scanner keyboard = new Scanner(System.in);

    numerator = keyboard.nextInt( );
    System.out.println("Enter denominator:");
    System.out.println("Be sure the denominator is not zero.");
    denominator = keyboard.nextInt( );

    if (denominator == 0)
    {
        System.out.println("I cannot do division by zero.");
        System.out.println("Since I cannot do what you want,");
        System.out.println("the program will now end.");
        System.exit(0);
    }

    quotient = ((double)numerator) / denominator;
    System.out.println(numerator + "/" + denominator +
                       " = " + quotient);
}
```

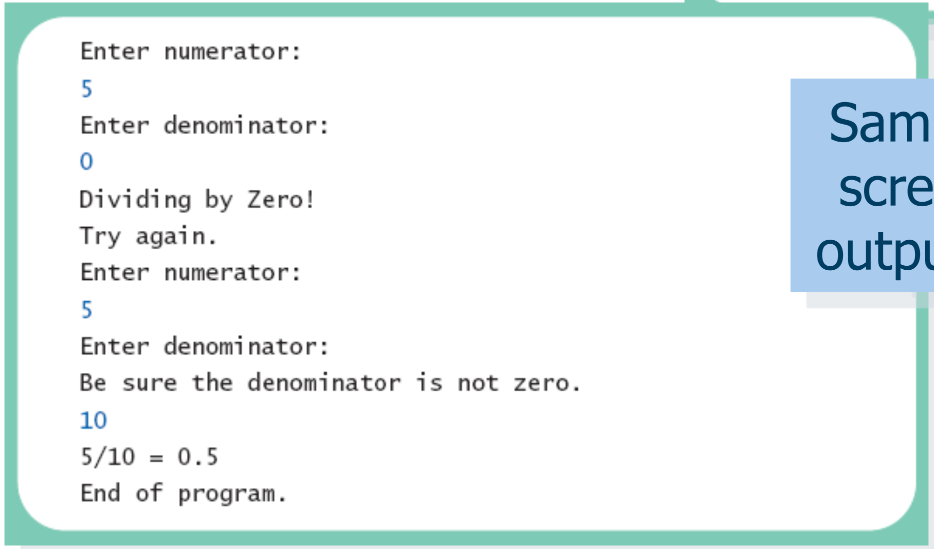
Defining Your Own Exception Classes

- Different runs of the program



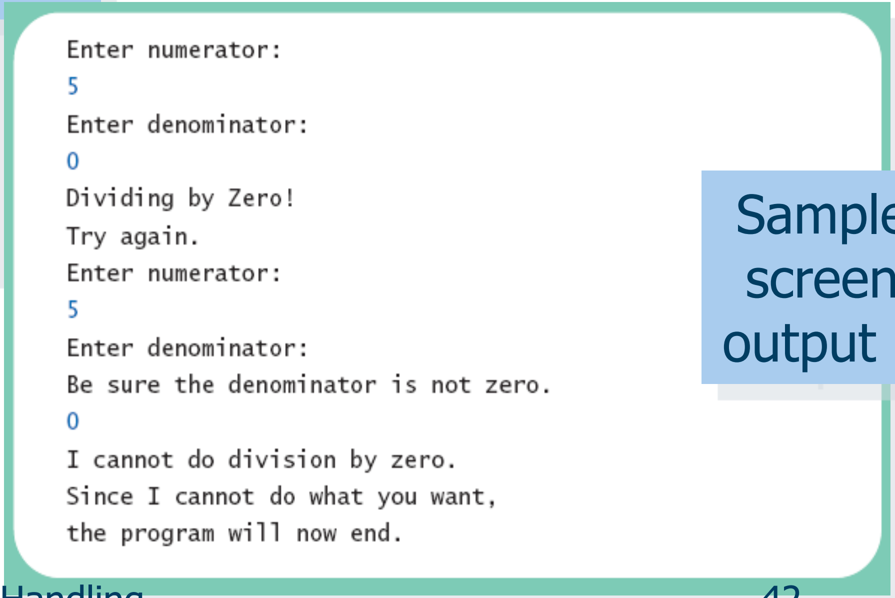
```
Enter numerator:
5
Enter denominator:
10
5/10 = 0.5
End of program.
```

Sample screen output 1



```
Enter numerator:
5
Enter denominator:
0
Dividing by Zero!
Try again.
Enter numerator:
5
Enter denominator:
Be sure the denominator is not zero.
10
5/10 = 0.5
End of program.
```

Sample screen output 2



```
Enter numerator:
5
Enter denominator:
0
Dividing by Zero!
Try again.
Enter numerator:
5
Enter denominator:
Be sure the denominator is not zero.
0
I cannot do division by zero.
Since I cannot do what you want,
the program will now end.
```

Sample screen output 3

Defining Your Own Exception Classes

- Note method `getMessage` defined in exception classes
  - Returns string passed as argument to constructor
  - If no actual parameter used, default message returned
- The type of an object is the name of the exception class

Defining Your Own Exception Classes

Guidelines

- ❑Use the `Exception` as the base class
- ❑Define at least one, but preferably two, constructors
  - Default, no parameter
  - With `String` parameter
- ❑Start constructor definition with call to constructor of base class, using `super`
- ❑No need to override inherited `getMessage`

Graphics Supplement

Graphics Supplement: Outline

- Exceptions in GUIs
- Programming Example: a `JFrame` GUI Using Exceptions

Exceptions in GUIs

- Not good practice to use `throws` clauses in the methods
  - In `JFrame` GUI or applet, uncaught exception **does not** end the program
  - However GUI may not cope correctly, user may receive sufficient instructions
- Thus most important to handle all checked exceptions correctly

Programming Example

- A `JFrame` GUI using exceptions
- View GUI class

  `class ColorDemo`
- Note exception class

  `class UnknownColorException`
- View driver program

  `class ShowColorDemo`

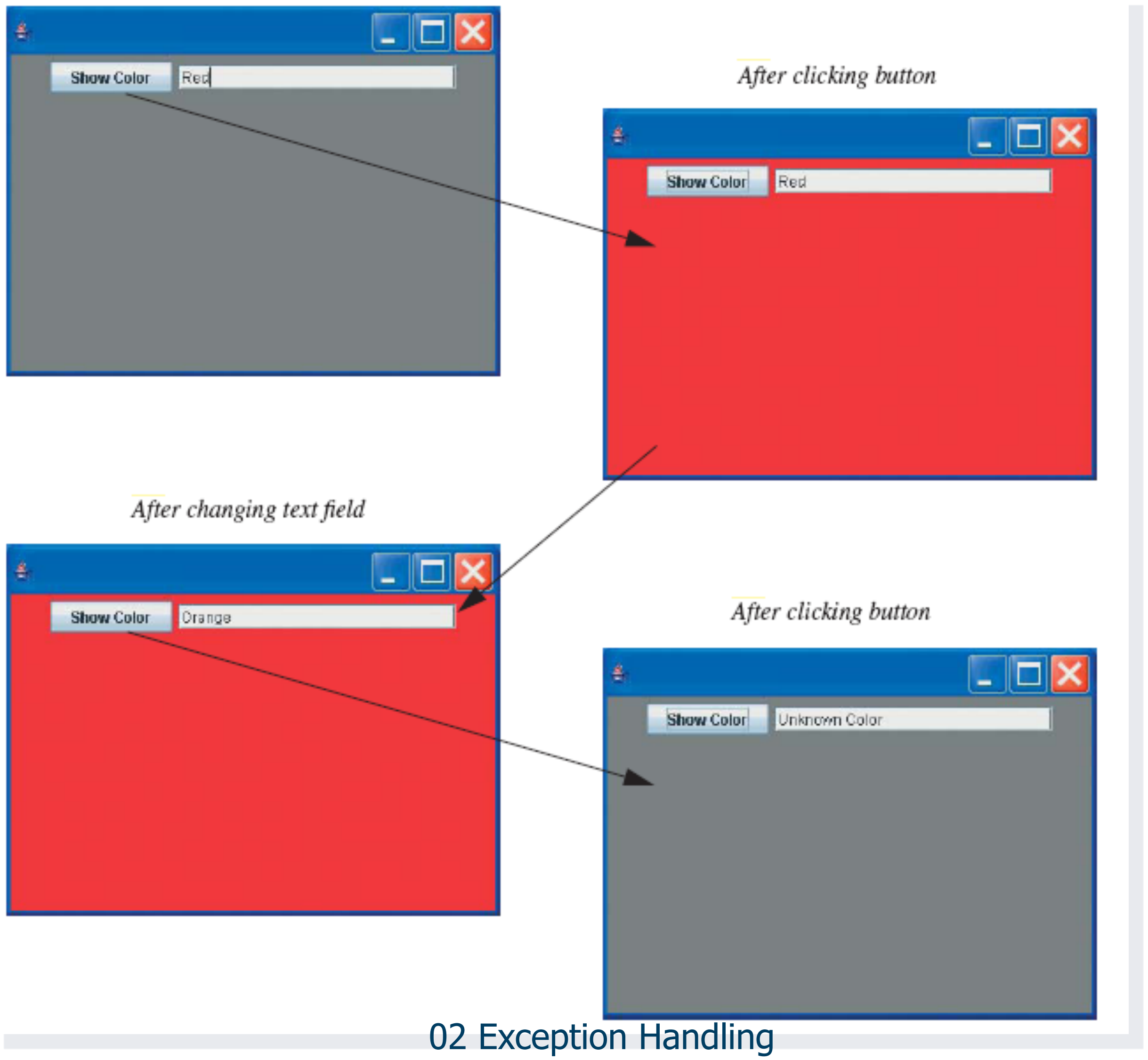
```java
public class ColorDemo extends JFrame implements ActionListener
{
    public static final int WIDTH = 400;
    public static final int HEIGHT = 300;
    public static final int NUMBER_OF_CHAR = 20;
    private JTextField colorName;

    public ColorDemo( ) {
        setSize(WIDTH, HEIGHT);
        WindowDestroyer listener = new WindowDestroyer( );
        addWindowListener(listener);

        Container contentPane = getContentPane( );
        contentPane.setBackground(Color.GRAY);
        contentPane.setLayout(new FlowLayout( ));

        JButton showButton = new JButton("Show Color");
        showButton.addActionListener(this);
        contentPane.add(showButton);

        colorName = new JTextField(NUMBER_OF_CHAR);
        contentPane.add(colorName);
    }

    public void actionPerformed(ActionEvent e){
        Container contentPane = getContentPane( );

        try {
            contentPane.setBackground(getColor(colorName.getText()));
        }
        catch(UnknownColorException exception) {
            colorName.setText("Unknown Color");
            contentPane.setBackground(Color.GRAY);
        }
    }

    public Color getColor(String name) throws UnknownColorException {
        if (name.equalsIgnoreCase("RED"))
            return Color.RED;
        else if (name.equalsIgnoreCase("WHITE"))
            return Color.WHITE;
        else if (name.equalsIgnoreCase("BLUE"))
            return Color.BLUE;
        else if (name.equalsIgnoreCase("GREEN"))
            return Color.GREEN;
        else
            throw new UnknownColorException();
    }
}
```

```java
/**
 If you register an object of this class as a listener to any
 object of the class JFrame, the object will end the program
 and close the JFrame, if the user clicks the JFrame's
 close-window button.
*/
public class WindowDestroyer extends WindowAdapter
{
    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}
```

```java
import ie.wit.impl.ColorDemo;

public class ShowColorDemo
{
    public static void main(String[] args)
    {
        ColorDemo gui = new ColorDemo( );
        gui.setVisible(true);
    }
}
```

```java
public class UnknownColorException extends Exception
{
    public UnknownColorException( )
    {
        super("Unknown Color!");
    }

    public UnknownColorException(String message)
    {
        super(message);
    }
}
```

Programming Example



Summary

- An exception is an object derived from class `Exception`
  - Descendants of class `Error` behave like exceptions
- Exception handling allows design of normal cases separate from exceptional situations
- Two kinds of exceptions
  - Checked and unchecked

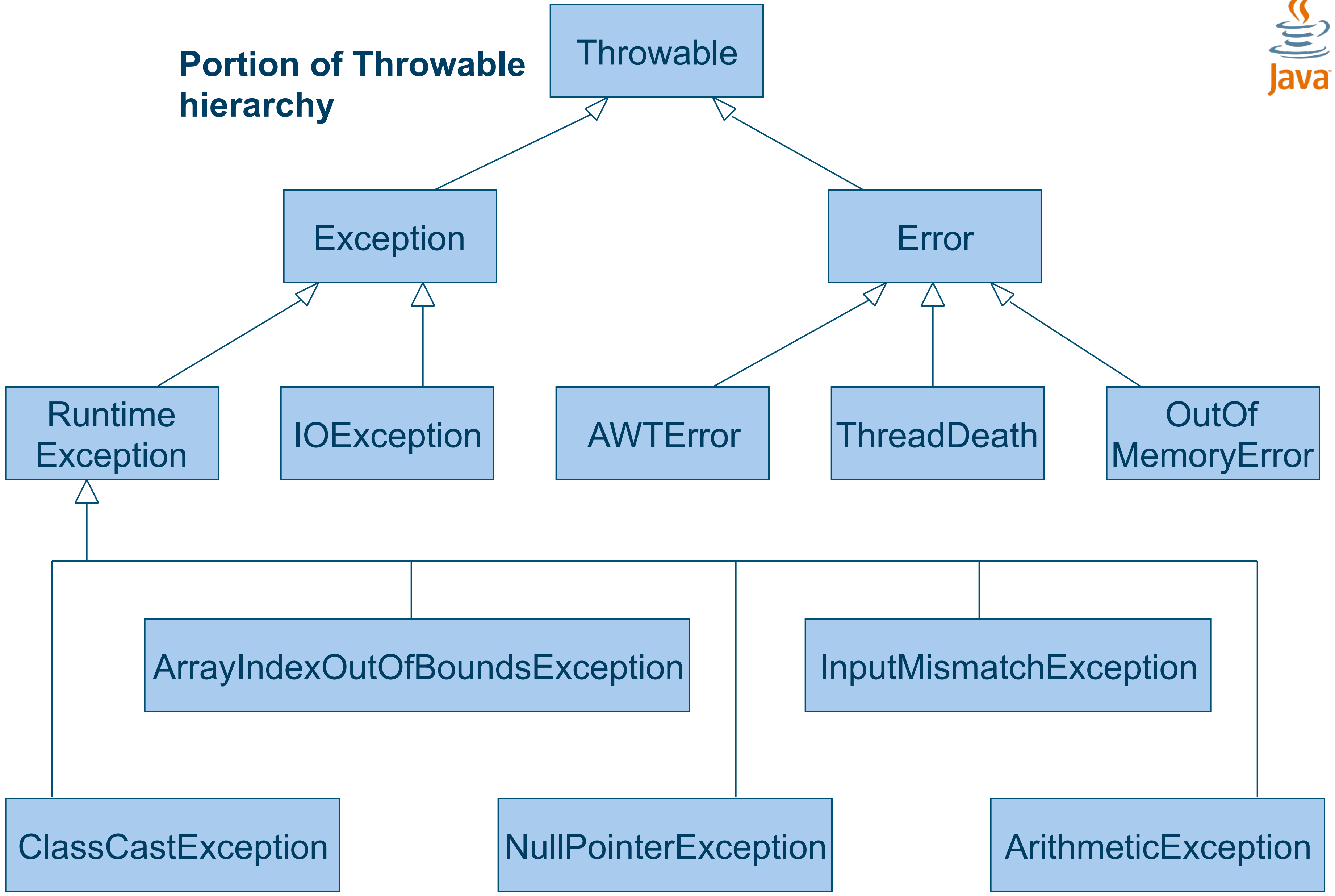
Summary

- Exceptions can be thrown by
  - Java statements
  - Methods from class libraries
  - Programmer use of `throw` statement
- Method that might `throw` but not `catch` an exception should use `throws` clause
- Exception is caught in `catch` block

Summary

- A `try` block followed by one or more `catch` blocks
  - More specific exception `catch` types should come first
- Every exception type has `getMessage` method usable to recover description of caught description
- Do not overuse exceptions

Portion of Throwable hierarchy



Questions?