# Chapter 9
# Exception Handling

Prof. Choonhwa Lee

Dept. of Computer Science and Engineering
Hanyang University

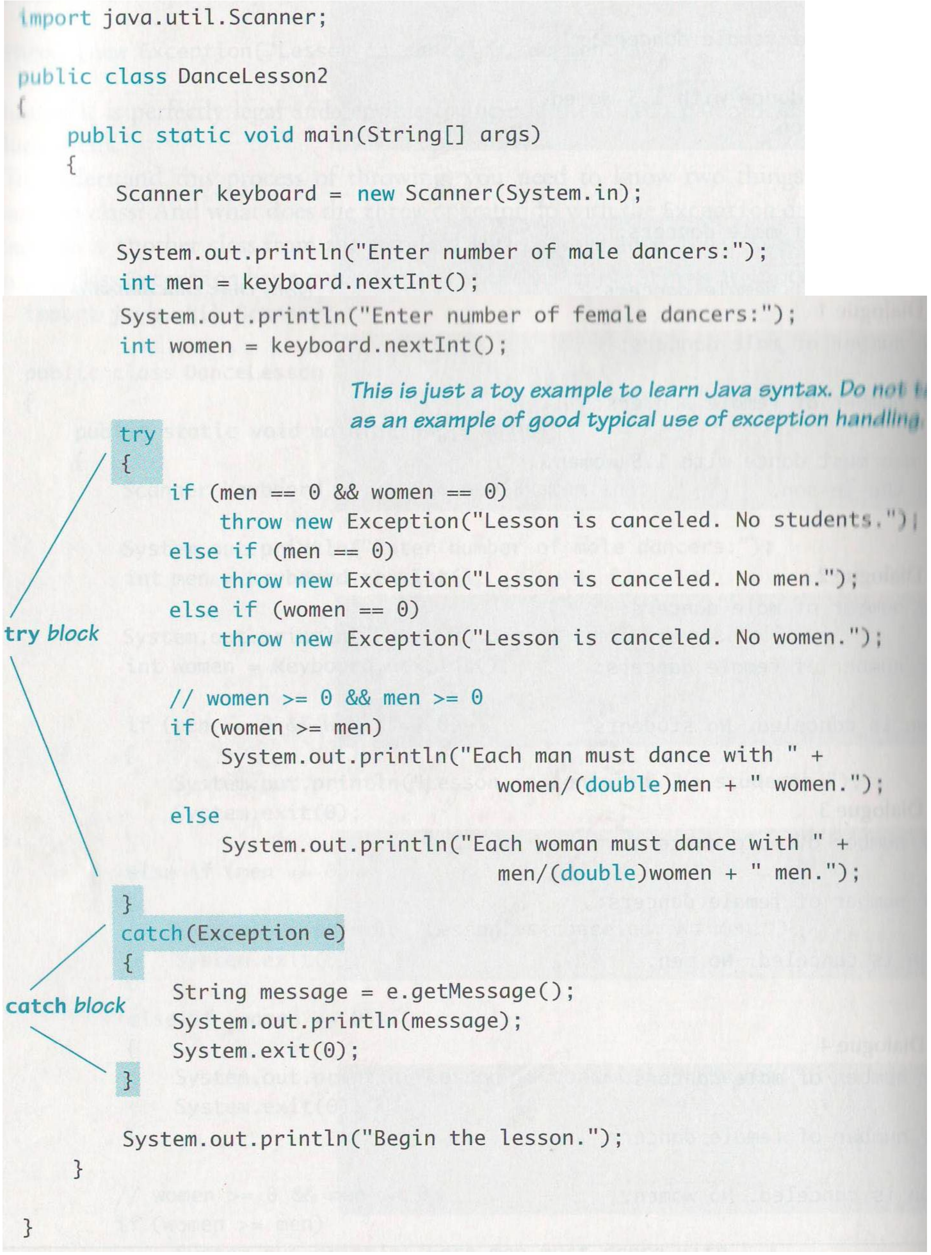
# Introduction to Exception Handling

- Sometimes the best outcome can be when nothing unusual happens
- However, the case where exceptional things happen must also be prepared for
  - Java exception handling facilities are used when the invocation of a method may cause something exceptional to occur

# Introduction to Exception Handling

- Java library software (or programmer-defined code) provides a mechanism that signals when something unusual happens
  - This is called *throwing an exception*
- In another place in the program, the programmer must provide code that deals with the exceptional case
  - This is called *handling the exception*

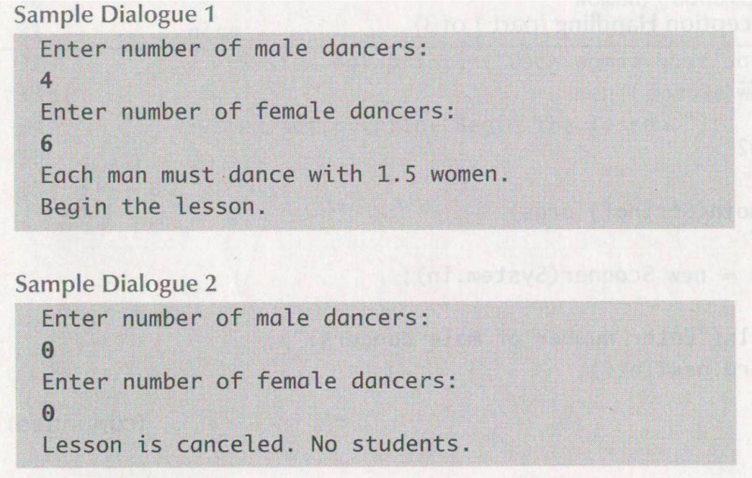
# Display 9.2 Same Thing Using Exception Handling



```
import java.util.Scanner;

public class DanceLesson2
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Enter number of male dancers:");
        int men = keyboard.nextInt();
        System.out.println("Enter number of female dancers:");
        int women = keyboard.nextInt();

        try
        {
            if (men == 0 && women == 0)
                throw new Exception("Lesson is canceled. No students.");
            else if (men == 0)
                throw new Exception("Lesson is canceled. No men.");
            else if (women == 0)
                throw new Exception("Lesson is canceled. No women.");

            // women >= 0 && men >= 0
            if (women >= men)
                System.out.println("Each man must dance with " +
                                   women/(double)men + " women.");
            else
                System.out.println("Each woman must dance with " +
                                   men/(double)women + " men.");
        }
        catch(Exception e)
        {
            String message = e.getMessage();
            System.out.println(message);
            System.exit(0);
        }

        System.out.println("Begin the lesson.");
    }
}
```

*This is just a toy example to learn Java syntax. Do not t*
*as an example of good typical use of exception handling*

try block
catch block



Sample Dialogue 1

```
Enter number of male dancers:
4
Enter number of female dancers:
6
Each man must dance with 1.5 women.
Begin the lesson.
```

Sample Dialogue 2

```
Enter number of male dancers:
0
Enter number of female dancers:
0
Lesson is canceled. No students.
```

# `try-throw-catch` Mechanism

- The basic way of handling exceptions in Java consists of the ***`try-throw-catch`*** trio
- The **`try`** block contains the code for the basic algorithm
  - It tells what to do when everything goes smoothly
- It is called a **`try`** block because it "tries" to execute the case where all goes as planned
  - It can also contain code that throws an exception if something unusual happens

```
try
{
   CodeThatMayThrowAnException
}
```

# **try-throw-catch** Mechanism

```
throw new
   ExceptionClassName(PossiblySomeArguments);
```

- When an exception is thrown, the execution of the surrounding **try** block is stopped
  - Normally, the flow of control is transferred to another portion of code known as the **catch** block
- The value thrown is the argument to the **throw** operator, and is always an object of some exception class
  - The execution of a **throw** statement is called *throwing an exception*

# `try-throw-catch` Mechanism

- A **`throw`** statement is similar to a method call:

```
throw new ExceptionClassName(SomeString);
```

  - In the above example, the object of class ***`ExceptionClassName`*** is created using a string as its argument
  - This object, which is an argument to the **`throw`** operator, is the exception object thrown
- Instead of calling a method, a **`throw`** statement calls a **`catch`** block

# **`try-throw-catch`** Mechanism

- When an exception is thrown, the **`catch`** block begins execution
  - The **`catch`** block has one parameter
  - The exception object thrown is plugged in for the **`catch`** block parameter
- The execution of the **`catch`** block is called *catching the exception*, or *handling the exception*
  - Whenever an exception is thrown, it should ultimately be handled (or caught) by some **`catch`** block

# `try-throw-catch` Mechanism

```
catch(Exception e)
{
  ExceptionHandlingCode
}
```

- A `catch` block looks like a method definition that has a parameter of type ***`Exception`*** class
  - It is not really a method definition, however
- A `catch` block is a separate piece of code that is executed when a program encounters and executes a `throw` statement in the preceding `try` block
  - A `catch` block is often referred to as an *exception handler*
  - It can have at most one parameter

# **`try-throw-catch`** Mechanism

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

- The identifier **`e`** in the above **`catch`** block heading is called the **`catch`** block parameter
- The **`catch`** block parameter does two things:

1. It specifies the type of thrown exception object that the **`catch`** block can catch (e.g., an **`Exception`** class object above)
2. It provides a name (for the thrown object that is caught) on which it can operate in the **`catch`** block
   - Note: The identifier ***`e`*** is often used by convention, but any non-keyword identifier can be used

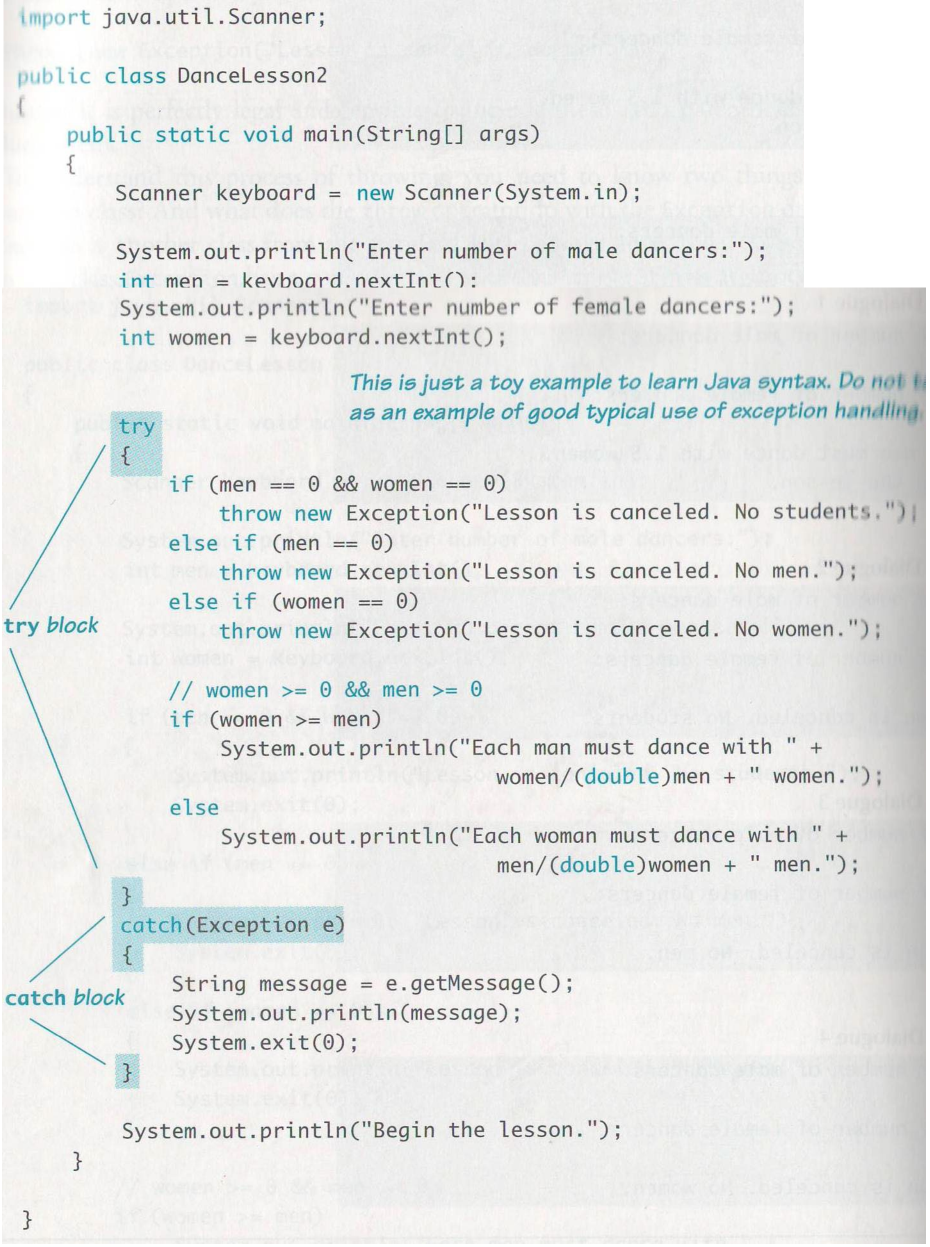
# `try-throw-catch` Mechanism

- When a `try` block is executed, two things can happen:
  1. No exception is thrown in the `try` block
     - The code in the `try` block is executed to the end of the block
     - The `catch` block is skipped
     - The execution continues with the code placed after the `catch` block

# `try-throw-catch` Mechanism

2. An exception is thrown in the `try` block and caught in the `catch` block
   - The rest of the code in the `try` block is skipped
   - Control is transferred to a following `catch` block (in simple cases)
   - The thrown object is plugged in for the `catch` block parameter
   - The code in the `catch` block is executed
   - The code that follows that `catch` block is executed (if any)

# Display 9.2 Same Thing Using Exception Handling

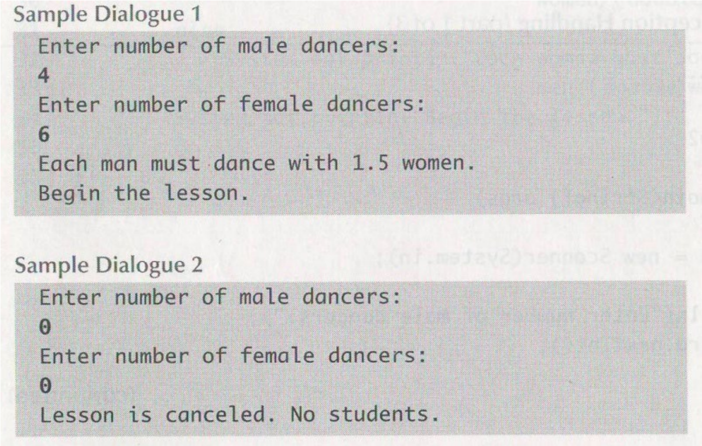


```java
import java.util.Scanner;

public class DanceLesson2
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);

        System.out.println("Enter number of male dancers:");
        int men = keyboard.nextInt();
        System.out.println("Enter number of female dancers:");
        int women = keyboard.nextInt();

        try
        {
            if (men == 0 && women == 0)
                throw new Exception("Lesson is canceled. No students.");
            else if (men == 0)
                throw new Exception("Lesson is canceled. No men.");
            else if (women == 0)
                throw new Exception("Lesson is canceled. No women.");

            // women >= 0 && men >= 0
            if (women >= men)
                System.out.println("Each man must dance with " +
                                   women/(double)men + " women.");
            else
                System.out.println("Each woman must dance with " +
                                   men/(double)women + " men.");
        }
        catch(Exception e)
        {
            String message = e.getMessage();
            System.out.println(message);
            System.exit(0);
        }

        System.out.println("Begin the lesson.");
    }
}
```

*This is just a toy example to learn Java syntax. Do not t*
*as an example of good typical use of exception handling.*

*try block*

*catch block*



Sample Dialogue 1

```
Enter number of male dancers:
4
Enter number of female dancers:
6
Each man must dance with 1.5 women.
Begin the lesson.
```

Sample Dialogue 2

```
Enter number of male dancers:
0
Enter number of female dancers:
0
Lesson is canceled. No students.
```

# Exception Classes

- There are more exception classes than just the single class **`Exception`**
  - There are more exception classes in the standard Java libraries
  - New exception classes can be defined like any other class
- All predefined exception classes have the following properties:
  - There is a constructor that takes a single argument of type **`String`**
  - The class has an accessor method **`getMessage`** that can recover the string given as an argument to the constructor when the exception object was created
- All programmer-defined classes should have the same properties

# Using the **getMessage** Method

```
. . . // method code
try
{
   . . .
   throw new Exception(StringArgument);
   . . .
}
catch(Exception e)
{
   String message = e.getMessage();
   System.out.println(message);
   System.exit(0);
}  . . .
```

# Using the **`getMessage`** Method

- Every exception has a **`String`** instance variable that contains some message
  - This string typically identifies the reason for the exception
- In the previous example, **`StringArgument`** is an argument to the **`Exception`** constructor
- This is the string used for the value of the string instance variable of exception **`e`**
  - Therefore, the method call **`e.getMessage()`** returns this string

# Exception Classes from Standard Packages

- Numerous predefined exception classes are included in the standard packages that come with Java
  - For example:

    ```
    IOException
    NoSuchMethodException
    FileNotFoundException
    ```

  - Many exception classes must be imported in order to use them

    ```
    import java.io.IOException;
    ```

# Exception Classes from Standard Packages

- The predefined exception class **`Exception`** is the root class for all exceptions
  - Every exception class is a descendent class of the class **`Exception`**
  - Although the **`Exception`** class can be used directly in a class or program, it is most often used to define a derived class
  - The class **`Exception`** is in the **`java.lang`** package, and so requires no **`import`** statement

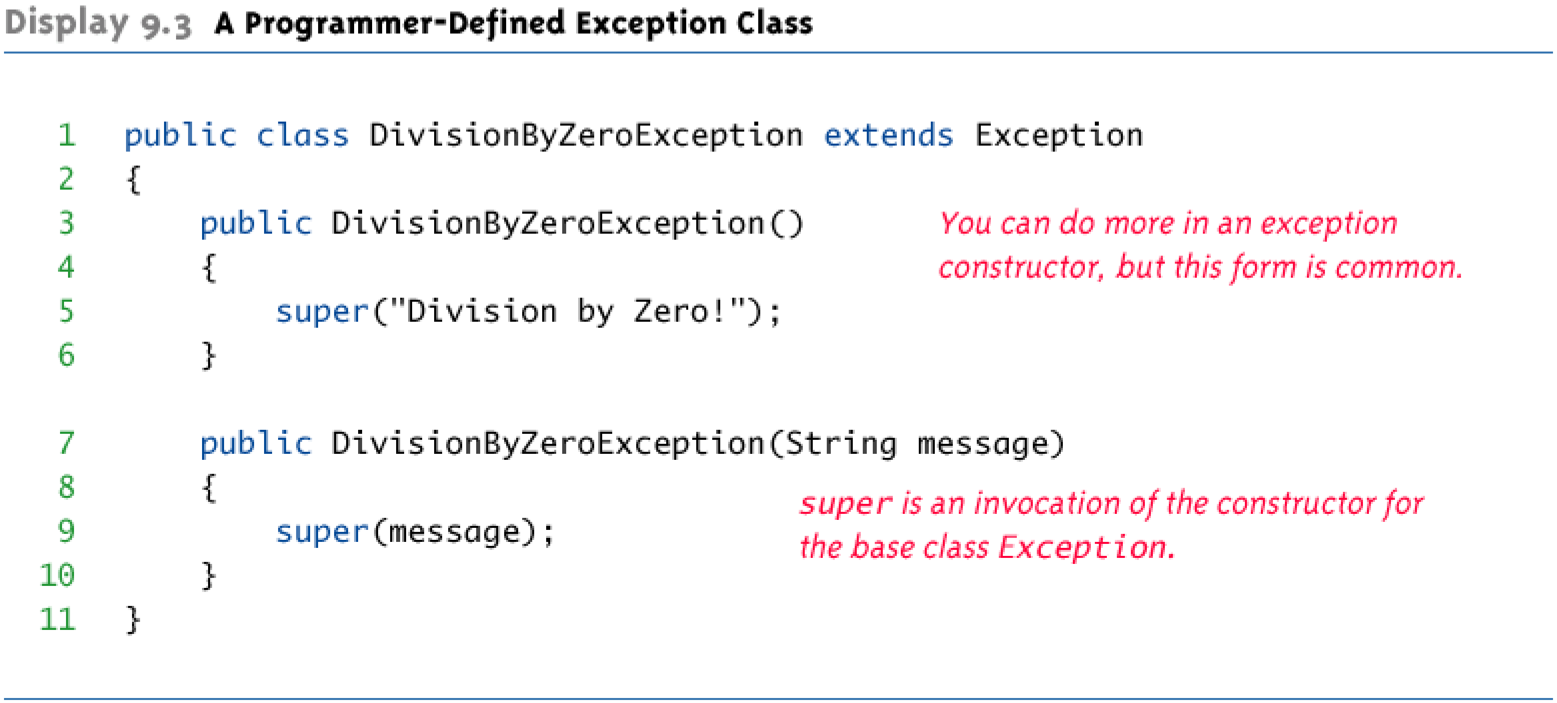
# Defining Exception Classes

- A **`throw`** statement can throw an exception object of any exception class
- Instead of using a predefined class,  exception classes can be programmer-defined
  - These can be tailored to carry the precise kinds of information needed in the **`catch`** block
  - A different type of exception can be defined to identify each different exceptional situation

# Defining Exception Classes

- Every exception class to be defined must be a derived class of some already defined exception class
  - It can be a derived class of any exception class in the standard Java libraries, or of any programmer defined exception class
- Constructors are the most important members to  define in an exception class
  - They must behave appropriately with respect to the variables and methods inherited from the base class
  - Often, there are no other members, except those inherited from the base class
- The following exception class performs these basic tasks only

# A Programmer-Defined Exception Class



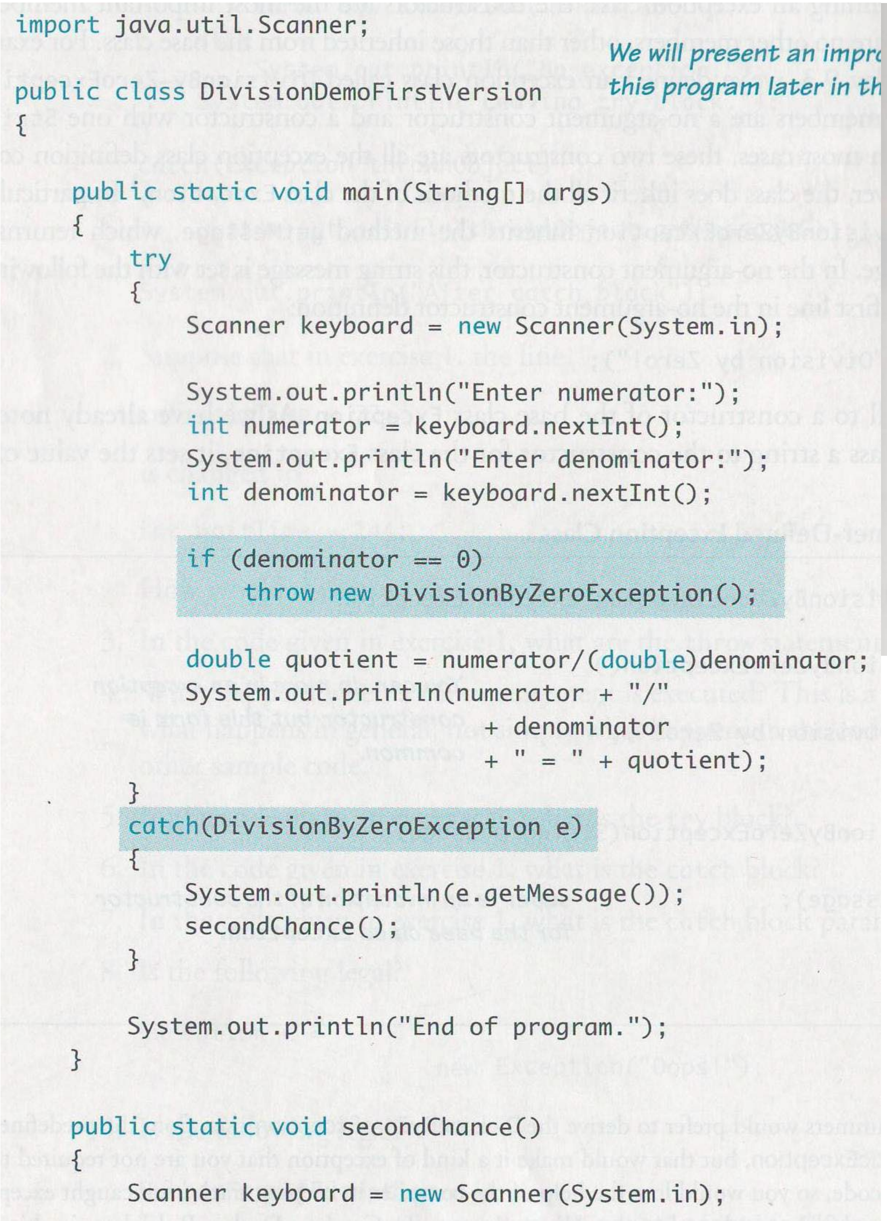
**Display 9.3 A Programmer-Defined Exception Class**

```java
public class DivisionByZeroException extends Exception
{
    public DivisionByZeroException()
    {
        super("Division by Zero!");
    }

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

*You can do more in an exception constructor, but this form is common.*

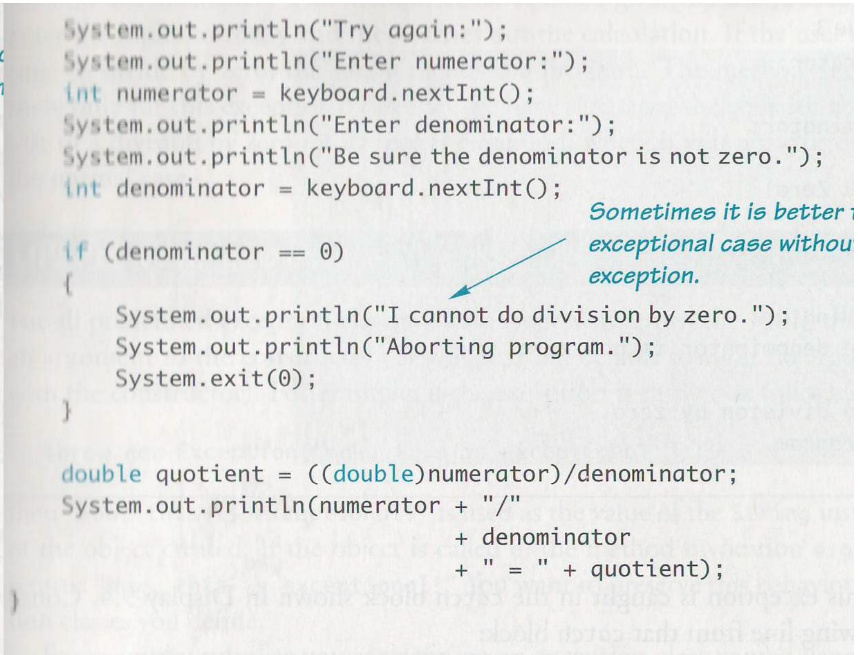
*super is an invocation of the constructor for the base class Exception.*

# Display 9.4 Using a Programmer-Defined Exception Class



```
import java.util.Scanner;

public class DivisionDemoFirstVersion
{

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

            System.out.println("Enter numerator:");
            int numerator = keyboard.nextInt();
            System.out.println("Enter denominator:");
            int denominator = keyboard.nextInt();

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

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

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

    public static void secondChance()
    {
        Scanner keyboard = new Scanner(System.in);
```

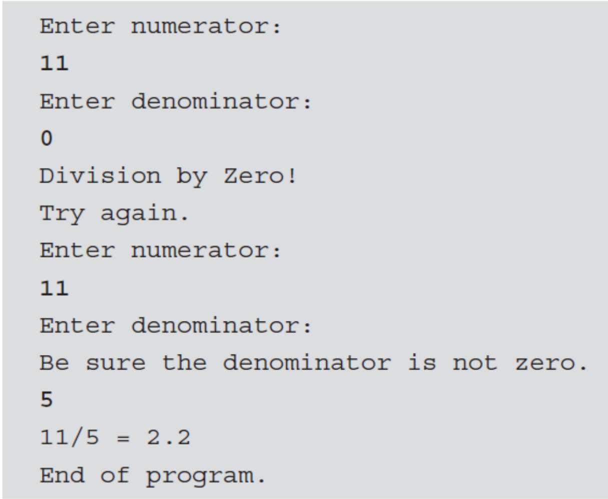
*We will present an impro*
*this program later in th*



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

        if (denominator == 0)
        {
            System.out.println("I cannot do division by zero.");
            System.out.println("Aborting program.");
            System.exit(0);
        }

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

*Sometimes it is better t*
*exceptional case withou*
*exception.*

Sample Dialogue 2



```
Enter numerator:
11
Enter denominator:
0
Division by Zero!
Try again.
Enter numerator:
11
Enter denominator:
Be sure the denominator is not zero.
5
11/5 = 2.2
End of program.
```

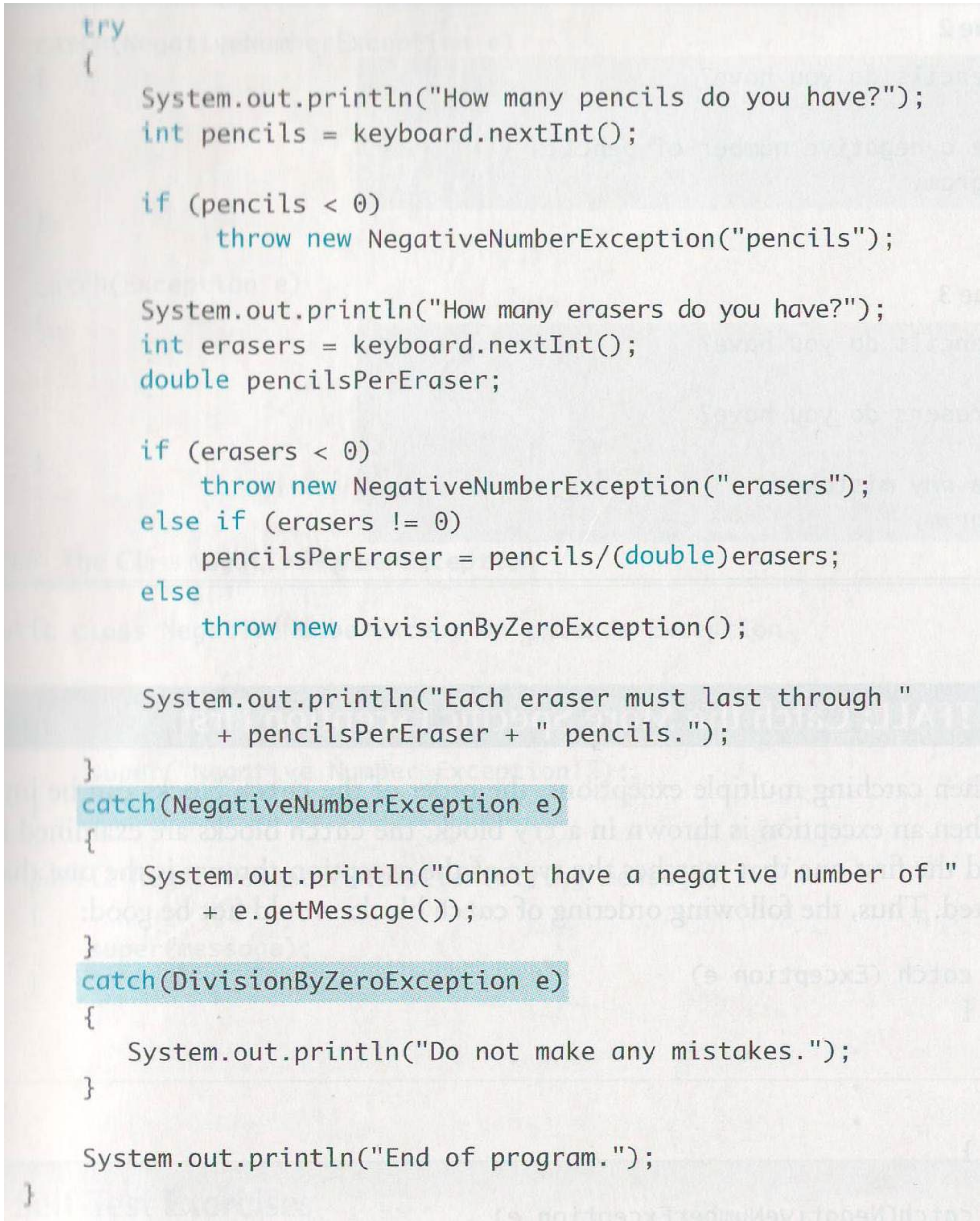
# Multiple **catch** Blocks

- A **try** block can potentially throw any number of exception values, and they can be of differing types
  - In any one execution of a **try** block, at most one exception can be thrown (since a throw statement ends the execution of the **try** block)
  - However, different types of exception values can be thrown on different executions of the **try** block

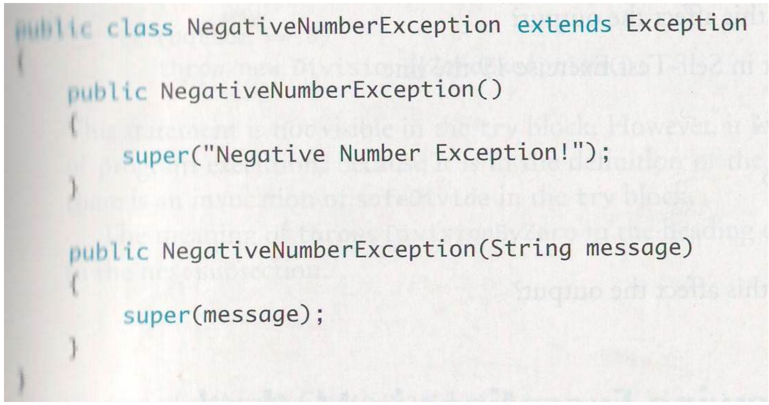
# Multiple **`catch`** Blocks

- Each **`catch`** block can only catch values of the exception class type given in the **`catch`** block heading
- Different types of exceptions can be caught by placing more than one **`catch`** block after a **`try`** block
  - Any number of **`catch`** blocks can be included, but they must be placed in the correct order

# Display 9.7 Catching Multiple Exception



```
    try
    {
        System.out.println("How many pencils do you have?");
        int pencils = keyboard.nextInt();

        if (pencils < 0)
             throw new NegativeNumberException("pencils");

        System.out.println("How many erasers do you have?");
        int erasers = keyboard.nextInt();
        double pencilsPerEraser;

        if (erasers < 0)
            throw new NegativeNumberException("erasers");
        else if (erasers != 0)
            pencilsPerEraser = pencils/(double)erasers;
        else
            throw new DivisionByZeroException();

        System.out.println("Each eraser must last through "
             + pencilsPerEraser + " pencils.");
    }
    catch(NegativeNumberException e)
    {
        System.out.println("Cannot have a negative number of "
            + e.getMessage());
    }
    catch(DivisionByZeroException e)
    {
       System.out.println("Do not make any mistakes.");
    }

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



```
public class NegativeNumberException extends Exception
{
    public NegativeNumberException()
    {
        super("Negative Number Exception!");
    }

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

# Pitfall: Catch the More Specific Exception First

- When catching multiple exceptions, the order of the `catch` blocks is important
  - When an exception is thrown in a `try` block, the `catch` blocks are examined in order
  - The first one that matches the type of the exception thrown is the one that is executed

# Pitfall: Catch the More Specific Exception First

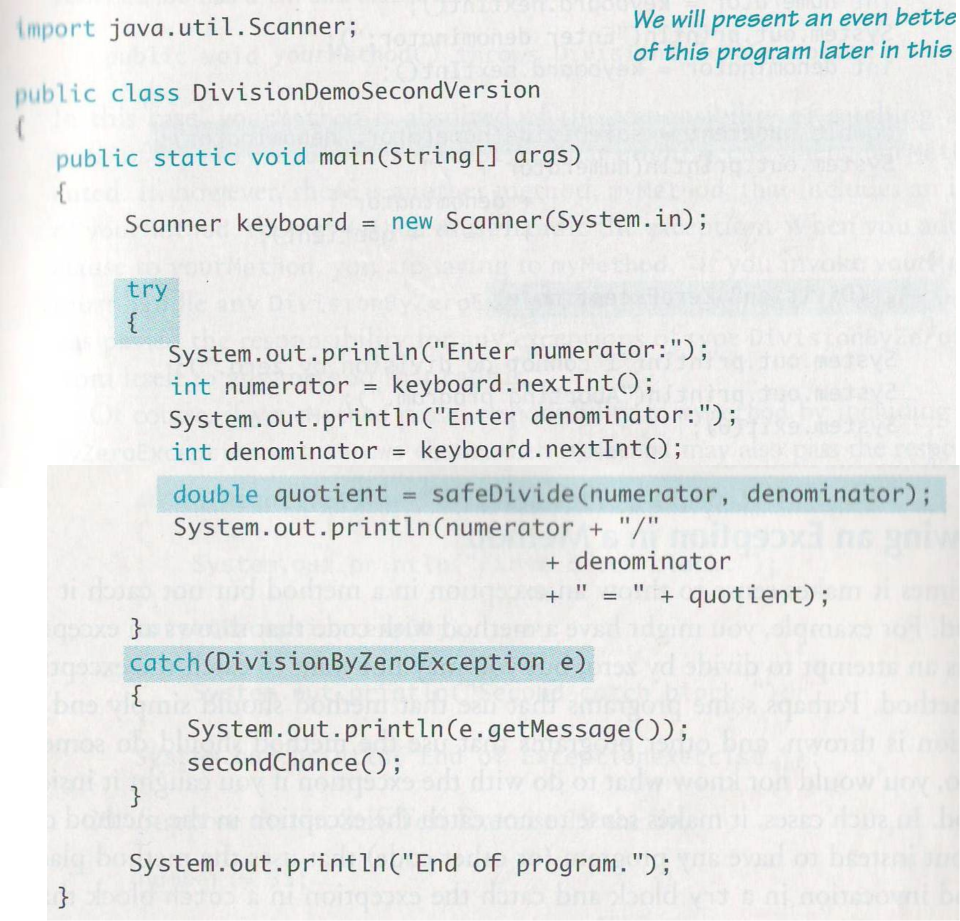
```
catch (Exception e)
{ . . . }
catch (NegativeNumberException e)
{ . . . }
```

- Because a **NegativeNumberException** is a type of **Exception**, all **NegativeNumberExceptions** will be caught by the first **catch** block before ever reaching the second block
  - The catch block for **NegativeNumberException** will never be used!
- For the correct ordering, simply reverse the two blocks

# Throwing an Exception in a Method

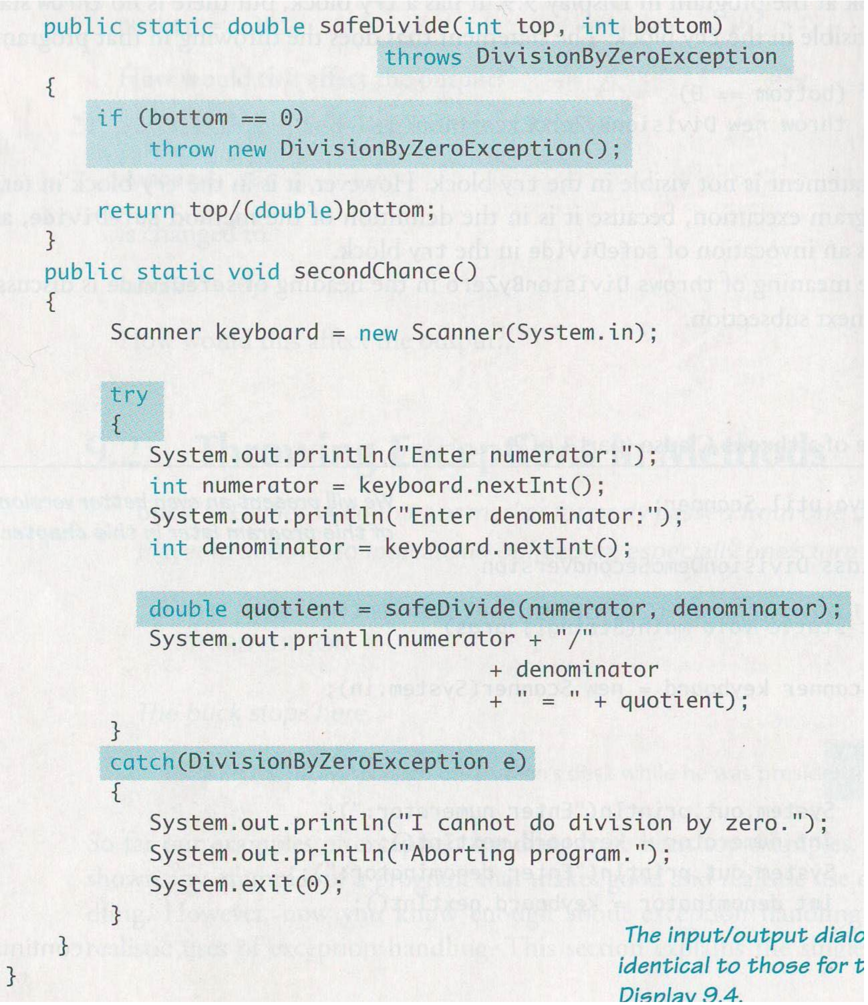
- Sometimes it makes sense to throw an exception in a method, but not catch it in the same method
  - Some programs that use a method should just end if an exception is thrown, and other programs should do something else
  - In such cases, the program using the method should  enclose the method invocation in a **`try`** block, and catch the exception in a **`catch`** block that follows
- In this case, the method itself would not include **`try`** and **`catch`** blocks
  - However, it would have to include a ***`throws`*** *clause*

# Display 9.9 Use of a throws Clause



```
import java.util.Scanner;

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

      try
      {
          System.out.println("Enter numerator:");
          int numerator = keyboard.nextInt();
          System.out.println("Enter denominator:");
          int denominator = keyboard.nextInt();

          double quotient = safeDivide(numerator, denominator);
          System.out.println(numerator + "/"
                                 + denominator
                                 + " = " + quotient);
      }
      catch(DivisionByZeroException e)
      {
          System.out.println(e.getMessage());
          secondChance();
      }

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

*We will present an even bette*
*of this program later in this*



```
   public static double safeDivide(int top, int bottom)
                        throws DivisionByZeroException
   {
       if (bottom == 0)
          throw new DivisionByZeroException();

       return top/(double)bottom;
   }
   public static void secondChance()
   {
       Scanner keyboard = new Scanner(System.in);

       try
       {
          System.out.println("Enter numerator:");
          int numerator = keyboard.nextInt();
          System.out.println("Enter denominator:");
          int denominator = keyboard.nextInt();

          double quotient = safeDivide(numerator, denominator);
          System.out.println(numerator + "/"
                                 + denominator
                                 + " = " + quotient);
       }
       catch(DivisionByZeroException e)
       {
          System.out.println("I cannot do division by zero.");
          System.out.println("Aborting program.");
          System.exit(0);
       }
   }
}
```

*The input/output dialo*
*identical to those for t*
*Display 9.4.*

# Declaring Exceptions in a **`throws`** Clause

- If a method can throw an exception but does not catch it, it must provide a warning
  - This warning is called a ***`throws`*** *clause*
  - The process of including an exception class in a throws clause is called *declaring the exception*

```
throws AnException  //throws clause
```

  - The following states that an invocation of **`aMethod`** could throw **`AnException`**

```
public void aMethod() throws AnException
```

# Declaring Exceptions in a **`throws`** Clause

- If a method can throw more than one type of exception, then separate the exception types by commas

```
public void aMethod() throws
   AnException, AnotherException
```

- If a method throws an exception and does not catch it, then the method invocation ends immediately

# The Catch or Declare Rule

- Most ordinary exceptions that might be thrown within a method must be accounted for in one of two ways:

1. The code that can throw an exception is placed within a **`try`** block, and the possible exception is caught in a **`catch`** block within the same method
2. The possible exception can be declared at the start of the method definition by placing the exception class name in a **`throws`** clause

# The Catch or Declare Rule

- The first technique handles an exception in a `catch` block
- The second technique is a way to shift the exception handling responsibility to the method that invoked the exception throwing method
- The invoking method must handle the exception, unless it too uses the same technique to "pass the buck"
- Ultimately, every exception that is thrown should eventually be caught by a `catch` block in some method that does not just declare the exception class in a `throws` clause

# Checked and Unchecked Exceptions

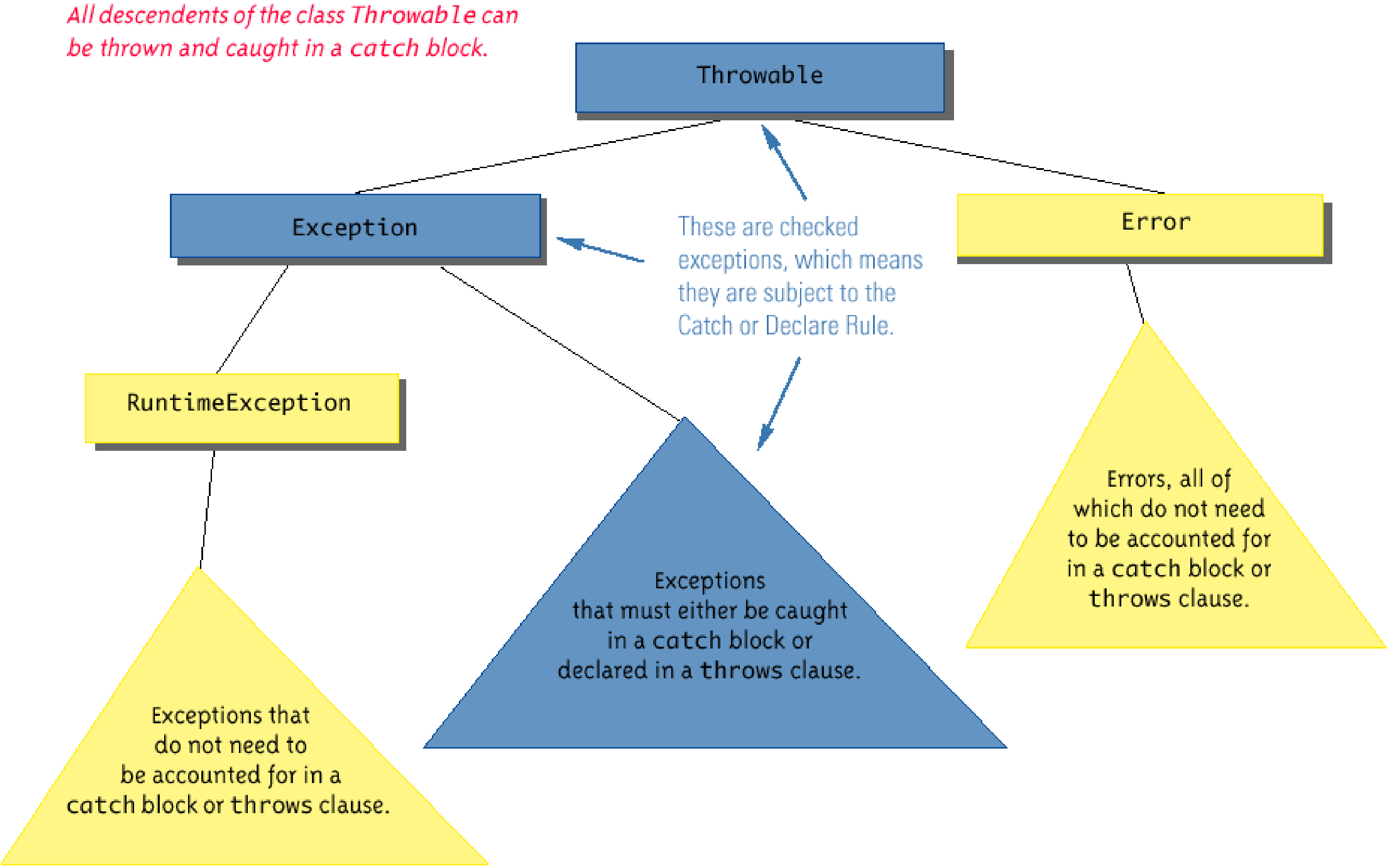
- Exceptions that are subject to the catch or declare rule are called *checked* exceptions
  - The compiler checks to see if they are accounted for with either a catch block or a throws clause
  - The classes **`Throwable`**, **`Exception`**, and all descendants of the class **`Exception`** are checked exceptions
- All other exceptions are *unchecked* exceptions
- The class **`Error`** and all its descendant classes are called *error classes*
  - Error classes are *not* subject to the Catch or Declare Rule

# Exceptions to the Catch or Declare Rule

- Checked exceptions must follow the Catch or Declare Rule
  - Programs in which these exceptions can be thrown will not compile until they are handled properly
- Unchecked exceptions are exempt from the Catch or Declare Rule
  - Programs in which these exceptions are thrown simply need to be corrected, as they result from some sort of error

# Hierarchy of Throwable Objects

**Display 9.10 Hierarchy of Throwable Objects**


*All descendents of the class Throwable can be thrown and caught in a catch block.*
Throwable
Exception
Error
These are checked exceptions, which means they are subject to the Catch or Declare Rule.
RuntimeException
Exceptions that do not need to be accounted for in a catch block or throws clause.
Exceptions that must either be caught in a catch block or declared in a throws clause.
Errors, all of which do not need to be accounted for in a catch block or throws clause.

# When to Use Exceptions

- Exceptions should be reserved for situations where a method encounters *an unusual or unexpected case that cannot be handled easily in some other way*
- When exception handling must be used, here are some basic guidelines:
  - Include **`throw`** statements and list the exception classes in a **`throws`** clause within a method definition
  - Place the **`try`** and **`catch`** blocks in a different method

# When to Use Exceptions

- Here is an example of a method from which the exception originates:

```
public void someMethod()
                         throws SomeException
{
   . . .
   throw new
             SomeException(SomeArgument);
   . . .
}
```

# When to Use Exceptions

- When `someMethod` is used by an `otherMethod`, the `otherMethod` must then deal with the exception:

```
 public void otherMethod()
 {
    try
    {
       someMethod();
       . . .
    }
    catch (SomeException e)
    {
       CodeToHandleException
    }
    . . .
 }
```

# The `finally` Block

- The `finally` block contains code to be executed whether or not an exception is thrown in a `try` block
  - If it is used, a `finally` block is placed after a `try` block and its following `catch` blocks

```
try
{   . . .   }
catch(ExceptionClass1 e)
{   . . .   }
   . . .
catch(ExceptionClassN e)
{   . . .   }
finally
{
   CodeToBeExecutedInAllCases
}
```

# The `finally` Block

- If the **`try-catch-finally`** blocks are inside a method definition, there are three possibilities when the code is run:

1. The **`try`** block runs to the end, no exception is thrown, and the finally block is executed
2. An exception is thrown in the **`try`** block, caught in one of the **`catch`** blocks, and the **`finally`** block is executed
3. An exception is thrown in the **`try`** block, there is no matching **`catch`** block in the method, the **`finally`** block is executed, and then the method invocation ends and the exception object is thrown to the enclosing method

# Exception Handling with the **`Scanner`** Class

- The **`nextInt`** method of the **`Scanner`** class can be used to read **`int`** values from the keyboard
- However, if a user enters something other than a well-formed **`int`** value, an **`InputMismatchException`** will be thrown
  - Unless this exception is caught, the program will end with an error message
  - If the exception is caught, the **`catch`** block can give code for some alternative action, such as asking the user to reenter the input

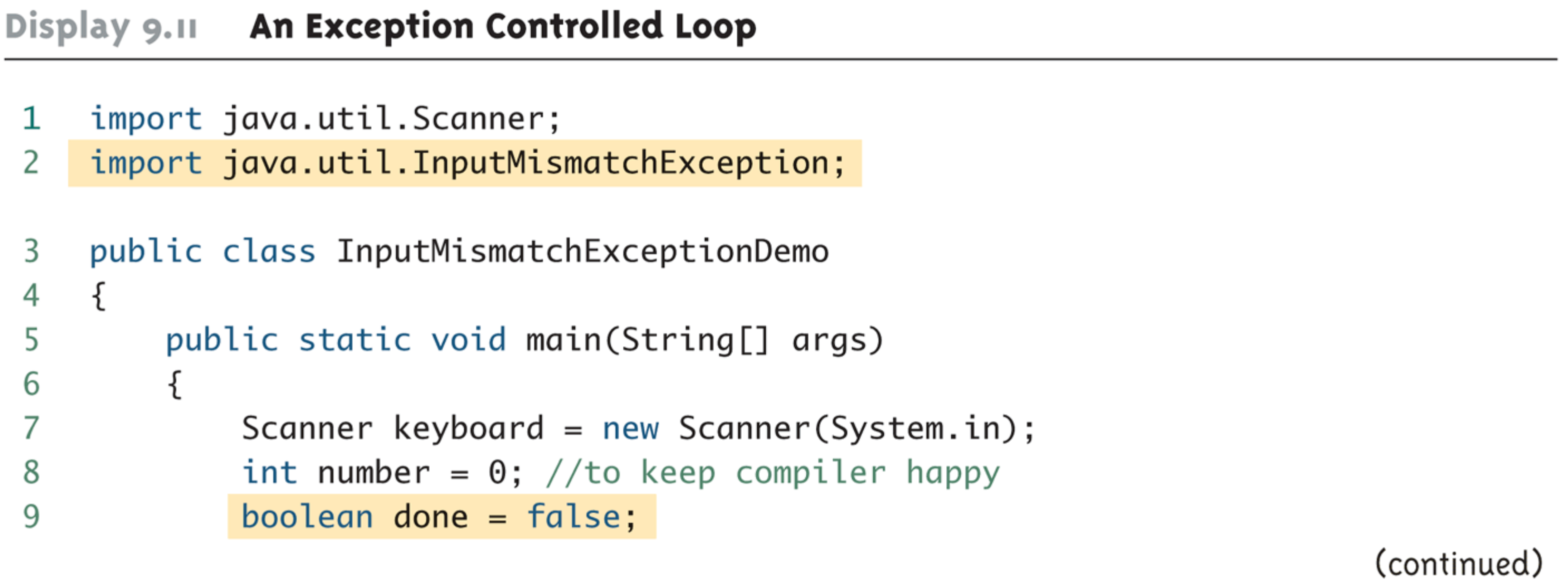
# The InputMismatchException

- The **InputMismatchException** is in the standard Java package **java.util**
  - A program that refers to it must use an **import** statement, such as the following:

```
import java.util.InputMismatchException;
```

- It is a descendent class of **RuntimeException**
  - Therefore, it is an unchecked exception and does not have to be caught in a **catch** block or declared in a **throws** clause
  - However, catching it in a **catch** block is allowed, and can sometimes be useful

# An Exception Controlled Loop (Part 1 of 3)

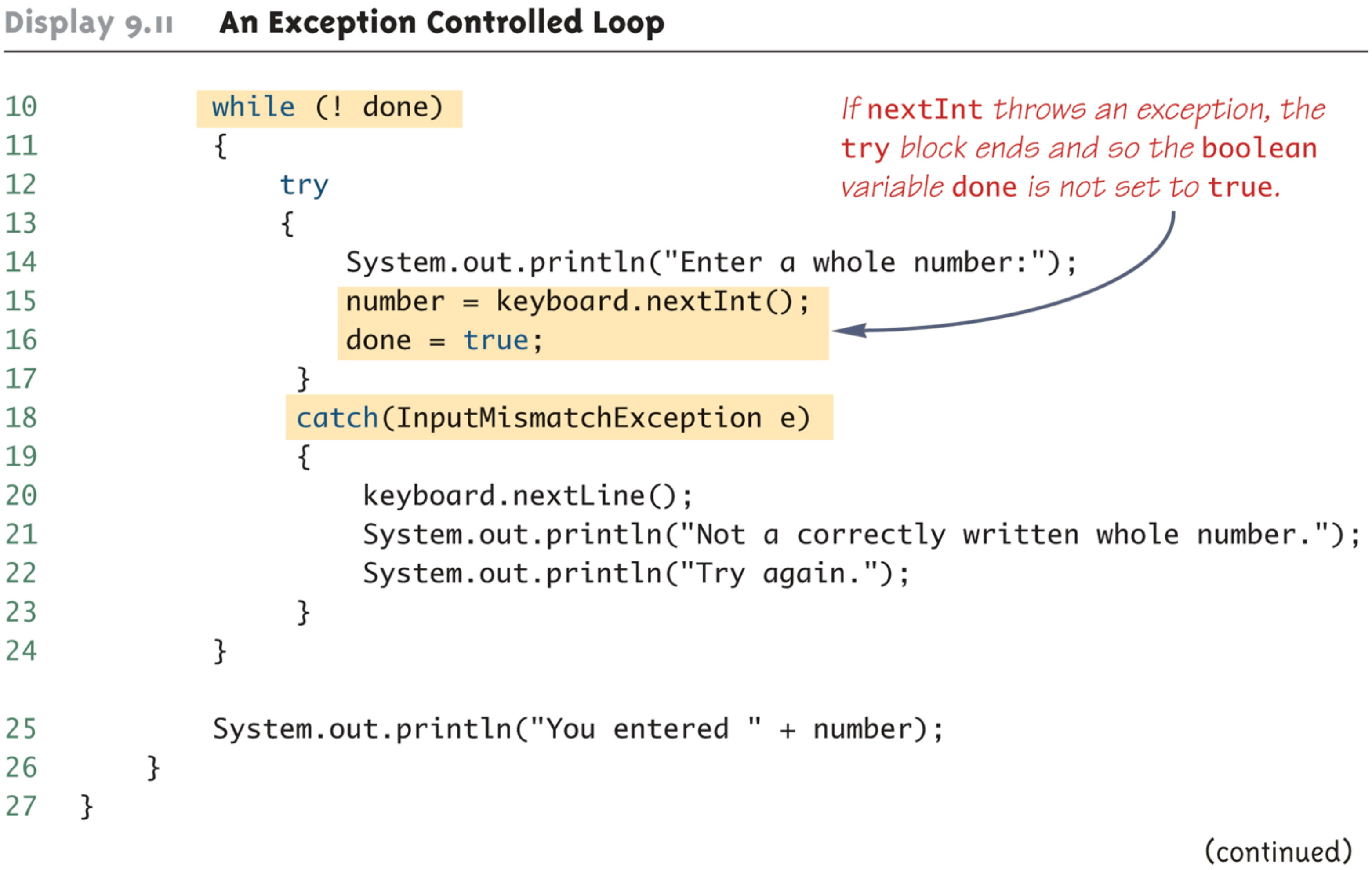


Display 9.11 **An Exception Controlled Loop**

```
import java.util.Scanner;
import java.util.InputMismatchException;

public class InputMismatchExceptionDemo
{
    public static void main(String[] args)
    {
        Scanner keyboard = new Scanner(System.in);
        int number = 0; //to keep compiler happy
        boolean done = false;
```

(continued)

# An Exception Controlled Loop (Part 2 of 3)



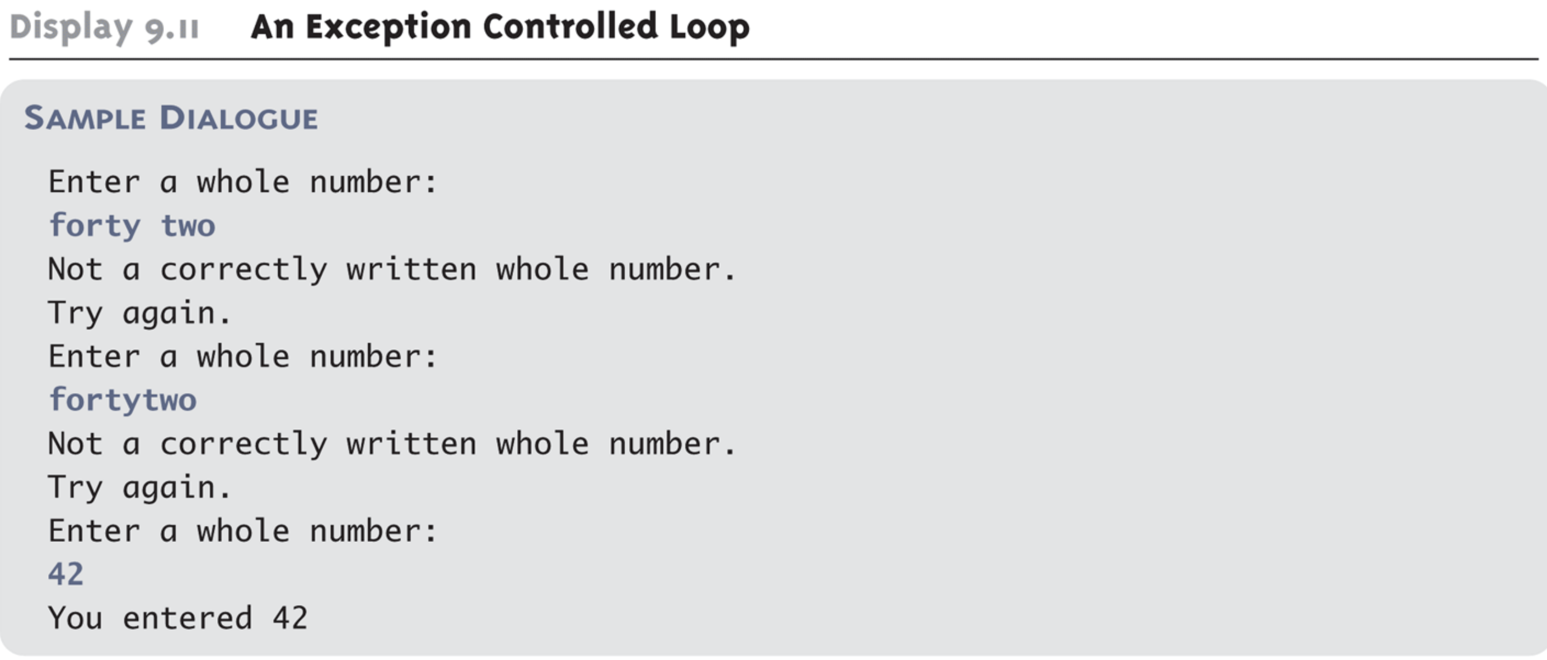
**Display 9.11 An Exception Controlled Loop**

```
        while (! done)
        {
            try
            {
                System.out.println("Enter a whole number:");
                number = keyboard.nextInt();
                done = true;
             }
             catch(InputMismatchException e)
             {
                 keyboard.nextLine();
                 System.out.println("Not a correctly written whole number.");
                 System.out.println("Try again.");
             }
        }

        System.out.println("You entered " + number);
    }
}
```

*If* **nextInt** *throws an exception, the* **try** *block ends and so the* **boolean** *variable* **done** *is not set to* **true**.

(continued)

# An Exception Controlled Loop (Part 3 of 3)



**Display 9.11** **An Exception Controlled Loop**

**Sample Dialogue**

```
Enter a whole number:
forty two
Not a correctly written whole number.
Try again.
Enter a whole number:
fortytwo
Not a correctly written whole number.
Try again.
Enter a whole number:
42
You entered 42
```