Sukkur IBA University
Merit - Quality - Excellence

CSC-150 - Object Oriented Programming

Semester II (Spring 2020)

Course Instructor(s): Dr. Sher Muhammad, Muhammad Atif, Khalid Hussain

Lab 08: Exception Handling

Objective(s):

1. What & Why Exception Handling

1: Introduction of Exception Handling

What is an Exception?

An Exception is an unwanted event that interrupts the normal flow of the program. When an exception occurs program execution gets terminated. In such cases we get a system generated error message. The good thing about exceptions is that they can be handled in Java. By handling the exceptions we can provide a meaningful message to the user about the issue rather than a system generated message, which may not be understandable to a user.

Why an exception occurs?

There can be several reasons that can cause a program to throw exception. For example: Opening a non-existing file in your program, Network connection problem, bad input data provided by user etc.

Error vs Exception

Error: An Error indicates serious problem that a reasonable application should not try to catch.
Exception: Exception indicates conditions that a reasonable application might try to catch.

Exception Handling

If an exception occurs, which has not been handled by programmer then program execution gets terminated and a system generated error message is shown to the user. For example look at the system generated exception below:

An exception generated by the system is given below

```
Exception in thread "main" java.lang.ArithmeticException: / by zero at
ExceptionDemo.main(ExceptionDemo.java:5)
```

```
ExceptionDemo : The class name
```

```
main : The method name
```

```
ExceptionDemo.java : The filename
```

```
java:5 : Line number
```

This message is not user friendly so a user will not be able to understand what went wrong. In order to let them know the reason in simple language, we handle exceptions. We handle such conditions and then prints a user friendly warning message to user, which lets them correct the error as most of the time exception occurs due to bad data provided by user.

try-catch Mechanism

The basic way of handling exceptions in Java consists of the try-throw-catch trio. At this point, we will start with only try and catch . The general setup consists of a try block followed by one or more catch blocks. First let's describe what a try block is. A try block has the following syntax:

```
try
{
  Some_Code_That_May_Throw_An_Exception
}
```

This try block contains the code for the basic algorithm that tells what to do when everything goes smoothly. It is called a try block because it "tries" to execute the case where all goes well.

Now, an exception can be "thrown" as a way of indicating that something unusual happened. For example, if our code tries to divide by zero, then an ArithmeticException object is thrown. Initially we will have existing Java classes do the throwing.

As the name suggests, when something is "thrown," something goes from one place to another place. In Java, what goes from one place to another is the flow of control as well as the exception object that is thrown. When an exception is thrown, the code in the surrounding try block stops executing and (normally) another portion of code, known as a catch block, begins execution. The catch block has a parameter, and the exception object thrown is plugged in for this catch block parameter. This executing of the catch block is called catching the exception or handling the exception. When an exception is thrown, it should ultimately be handled by (caught by) some catch block. The appropriate catch block immediately follows the try block; for example:

```
catch(Exception e)
{
   String message = e.getMessage();
   System.out.println(message);
   System.exit(0);
}
```

This catch block looks very much like a method definition that has a parameter of a type Exception. By using the type Exception, this catch block will catch any possible exception that is thrown. We can also restrict the catch block to specific exception classes. The catch block is not a method definition, but in some ways, it is like a method. It is a separate piece of code that is executed when your code throws an exception. The catch block in the previous example will print out a message about the exception that was thrown.

So, when an exception is thrown, it is similar to a method call, but instead of calling a method, it calls the catch block and says to execute the code in the catch block.

A catch block is often referred to as an exception handler.

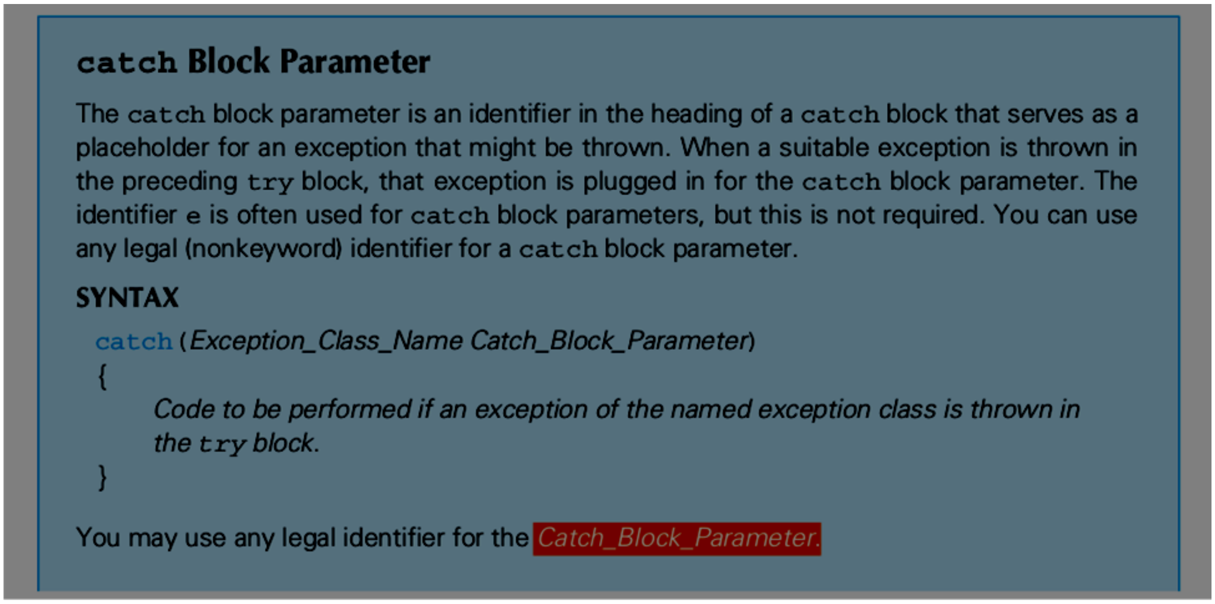
Let’s focus on the identifier e in the following line from a catch block:

```
catch(Exception e)
```

That identifier e in the catch block heading is called the catch block parameter .Each catch block can have at most one catch block parameter. The catch block parameter does two things:

- The catch block parameter is preceded by an exception class name that specifies what type of thrown exception object the catch block can catch. If the class name is Exception, then the block can catch any exception.
- The catch block parameter gives you a name for the thrown object that is caught, so you can write code in the catch block that does things with the thrown object that is caught.

Although the identifier e is often used for the catch block parameter, this is not required. You may use any no keyword identifier for the catch block parameter just as you can for a method parameter.



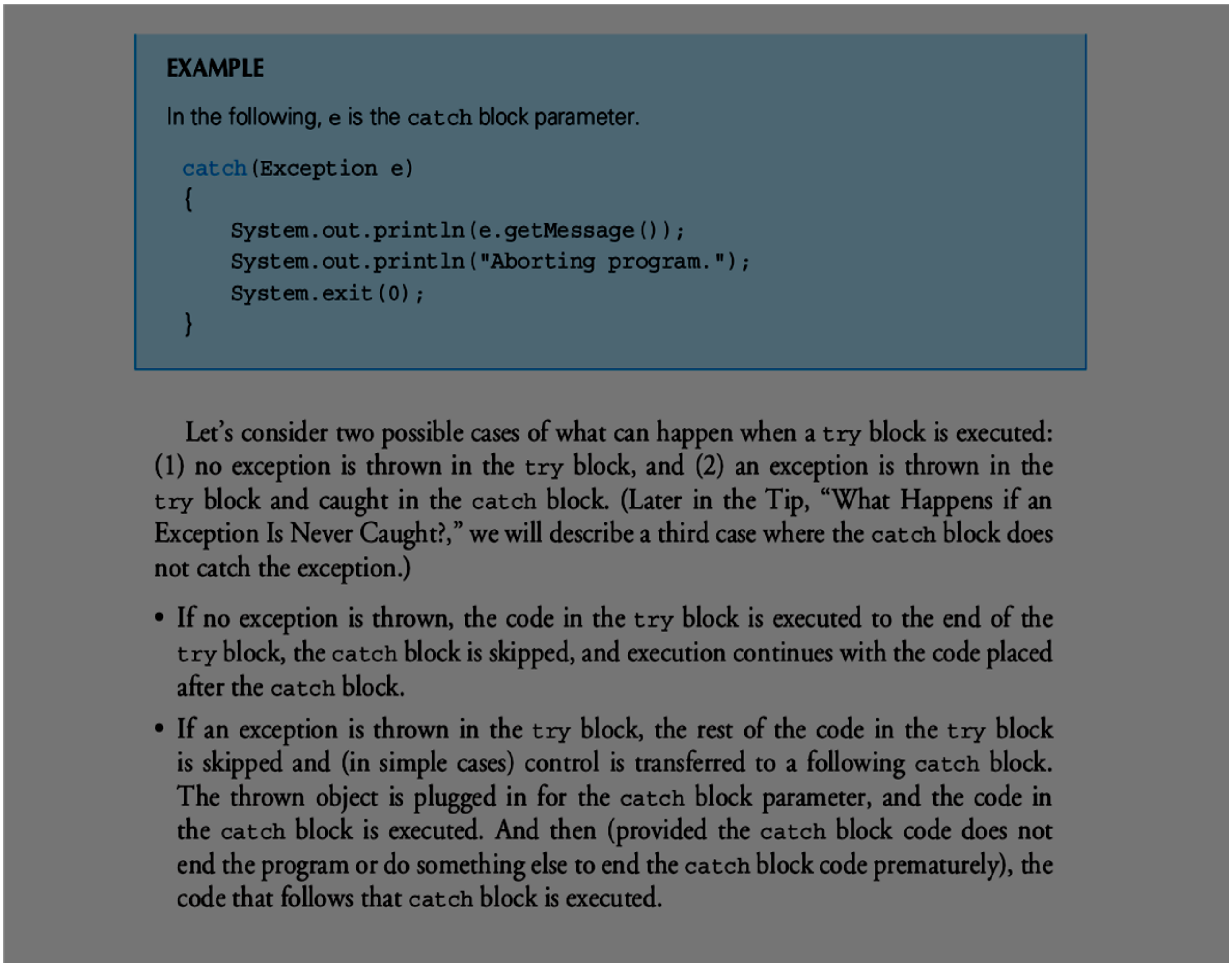
`catch` Block Parameter

The `catch` block parameter is an identifier in the heading of a `catch` block that serves as a placeholder for an exception that might be thrown. When a suitable exception is thrown in the preceding `try` block, that exception is plugged in for the `catch` block parameter. The identifier `e` is often used for `catch` block parameters, but this is not required. You can use any legal (nonkeyword) identifier for a `catch` block parameter.

SYNTAX

`catch` (*Exception_Class_Name Catch_Block_Parameter*)
{
Code to be performed if an exception of the named exception class is thrown in the `try` *block.*
}

You may use any legal identifier for the *Catch_Block_Parameter.*



EXAMPLE

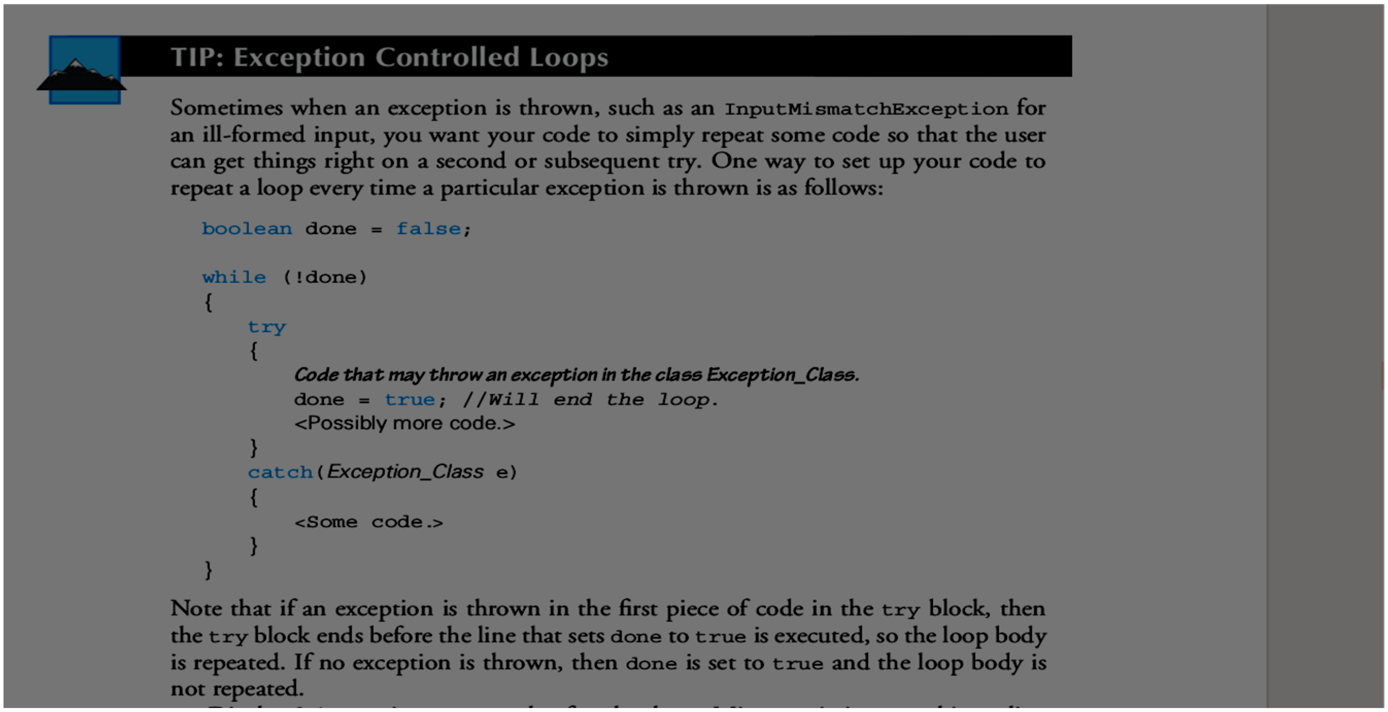
In the following, e is the catch block parameter.

```
catch(Exception e)
{
    System.out.println(e.getMessage());
    System.out.println("Aborting program.");
    System.exit(0);
}
```

Let's consider two possible cases of what can happen when a try block is executed: (1) no exception is thrown in the try block, and (2) an exception is thrown in the try block and caught in the catch block. (Later in the Tip, "What Happens if an Exception Is Never Caught?," we will describe a third case where the catch block does not catch the exception.)

- If no exception is thrown, the code in the try block is executed to the end of the try block, the catch block is skipped, and execution continues with the code placed after the catch block.
- If an exception is thrown in the try block, the rest of the code in the try block is skipped and (in simple cases) control is transferred to a following catch block. The thrown object is plugged in for the catch block parameter, and the code in the catch block is executed. And then (provided the catch block code does not end the program or do something else to end the catch block code prematurely), the code that follows that catch block is executed.

Example with Loop



TIP: Exception Controlled Loops

Sometimes when an exception is thrown, such as an InputMismatchException for an ill-formed input, you want your code to simply repeat some code so that the user can get things right on a second or subsequent try. One way to set up your code to repeat a loop every time a particular exception is thrown is as follows:

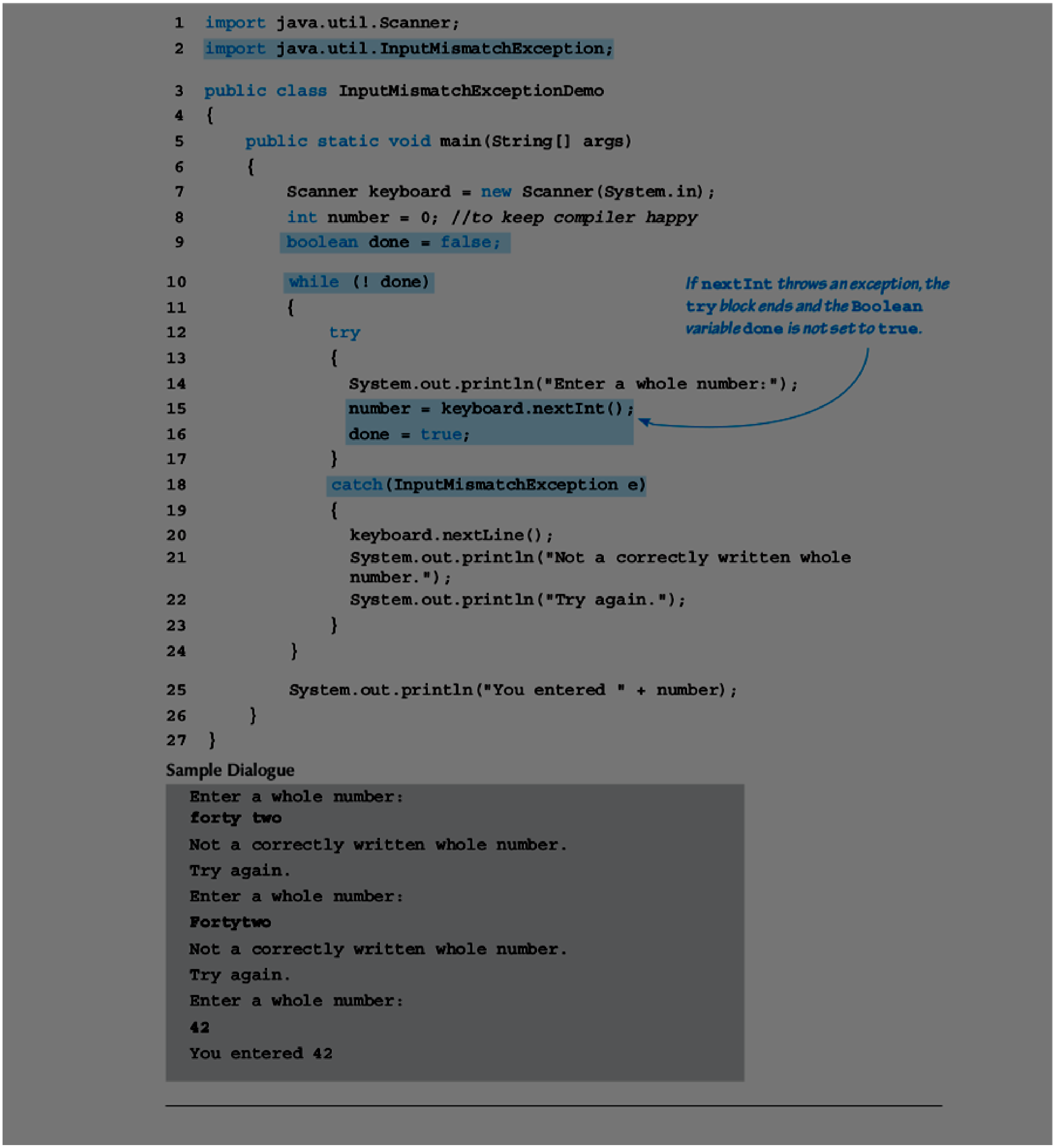
```
boolean done = false;

while (!done)
{
    try
    {
        Code that may throw an exception in the class Exception_Class.
        done = true; //Will end the loop.
        <Possibly more code.>
    }
    catch(Exception_Class e)
    {
        <Some code.>
    }
}
```

Note that if an exception is thrown in the first piece of code in the try block, then the try block ends before the line that sets done to true is executed, so the loop body is repeated. If no exception is thrown, then done is set to true and the loop body is not repeated.

Programming Example

Below Program contains an example of such a loop. Minor variations on this outline can accommodate a range of different situations for which you want to repeat code on throwing an exception.



```
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;

        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 the* **Boolean** *variable* **done** *is not set to* **true.**

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
```

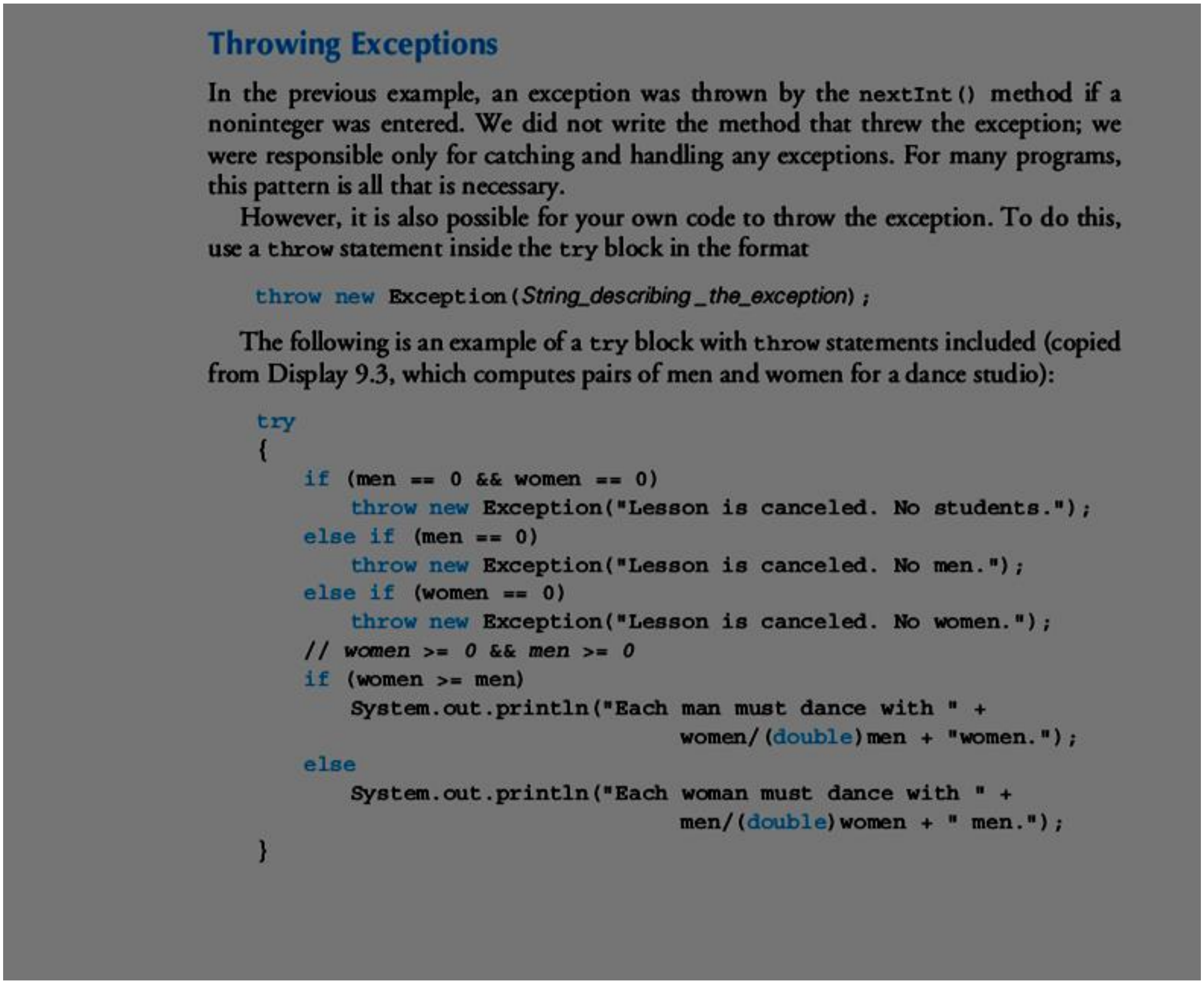
Exercise 1:

1. How would the dialogue in above given example program change if you were to omit the following line from the catch block? (Try it and see.)

```
keyboard.nextLine();
```

2. Give the definition for the following method. Use the techniques given in the above given program

```
/**
Precondition: keyboard is an object of the class Scanner that has been set up for keyboard input (as we have been doing
right along). Returns: An int value entered at the keyboard. If the user enters an incorrectly formed input, she or he
is prompted to reenter the value,
*/
public static int getInt(Scanner keyboard)
```



Throwing Exceptions

In the previous example, an exception was thrown by the `nextInt()` method if a noninteger was entered. We did not write the method that threw the exception; we were responsible only for catching and handling any exceptions. For many programs, this pattern is all that is necessary.

However, it is also possible for your own code to throw the exception. To do this, use a `throw` statement inside the `try` block in the format

```
throw new Exception(String_describing_the_exception);
```

The following is an example of a `try` block with `throw` statements included (copied from Display 9.3, which computes pairs of men and women for a dance studio):

```
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.");
}
```

throw statement

This `try` block contains the following three `throw` statements:

```
throw new Exception("Lesson is canceled. No students.");
throw new Exception("Lesson is canceled. No men.");
throw new Exception("Lesson is canceled. No women.");
```

throwing an exception

The value thrown is an 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.*

`throw` Statement

SYNTAX

`throw new` *Exception_Class_Name (Possibly_Some_Arguments);*

When the `throw` statement is executed, the execution of the surrounding `try` block is stopped and (normally) control is transferred to a `catch` block. The code in the `catch` block is executed next. See the box entitled `"try-throw-catch"` later in this chapter for more details.

EXAMPLE

```
throw new Exception("Division by zero.");
```

The `getMessage` Method

Every exception has a `String` instance variable that contains some message, which typically identifies the reason for the exception. For example, if the exception is thrown as follows,

`throw new Exception(`*String_Argument*`);`

then the string given as an argument to the constructor `Exception` is used as the value of this `String` instance variable. If the object is called `e`, then the method call `e.getMessage()` returns this string.

EXAMPLE

Suppose the following `throw` statement is executed in a `try` block:

```
throw new Exception("Input must be positive.");
```

And suppose the following is a `catch` block immediately following the `try` block:

```
catch (Exception e)
{
    System.out.println(e.getMessage());
    System.out.println("Program aborted.");
    System.exit(0);
}
```

In this case, the method call `e.getMessage()` returns the string
`"Input must be positive."`

Example Program without Exception

```
import java.util.Scanner;

public class DanceLesson

{

   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();

     if (men == 0 && women == 0){

         System.out.println("Lesson is canceled. No students.");

         System.exit(0);

     }

     else if (men == 0)

     {

         System.out.println("Lesson is canceled. No men.");

         System.exit(0);

     }

     else if (women == 0){

         System.out.println("Lesson is canceled. No women.");

         System.exit(0);

     }

    // 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.");

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

}}
```

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.
```

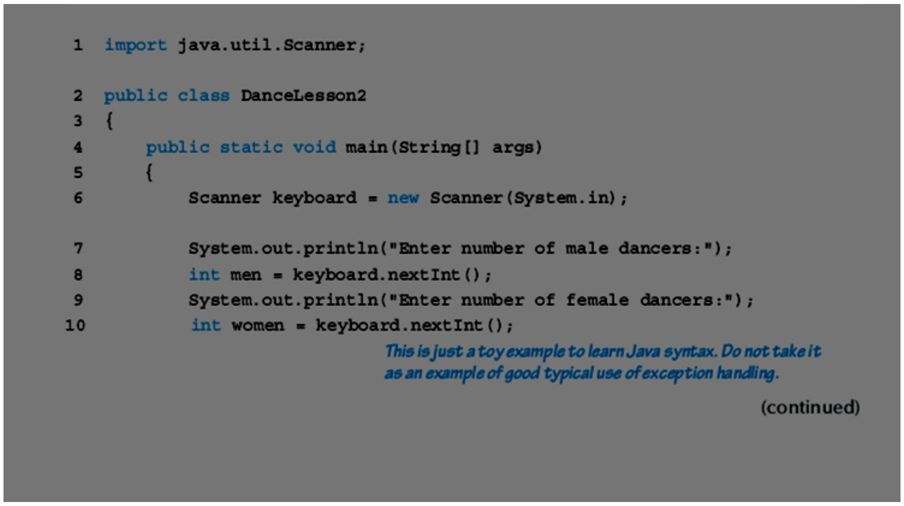
Sample Dialogue 3

```
Enter number of male dancers:
0
Enter number of female dancers:
5
Lesson is canceled. No men.
```

Sample Dialogue 4

```
Enter number of male dancers:
4
Enter number of female dancers:
0
Lesson is canceled. No women.
```

Same thing with Exception handling



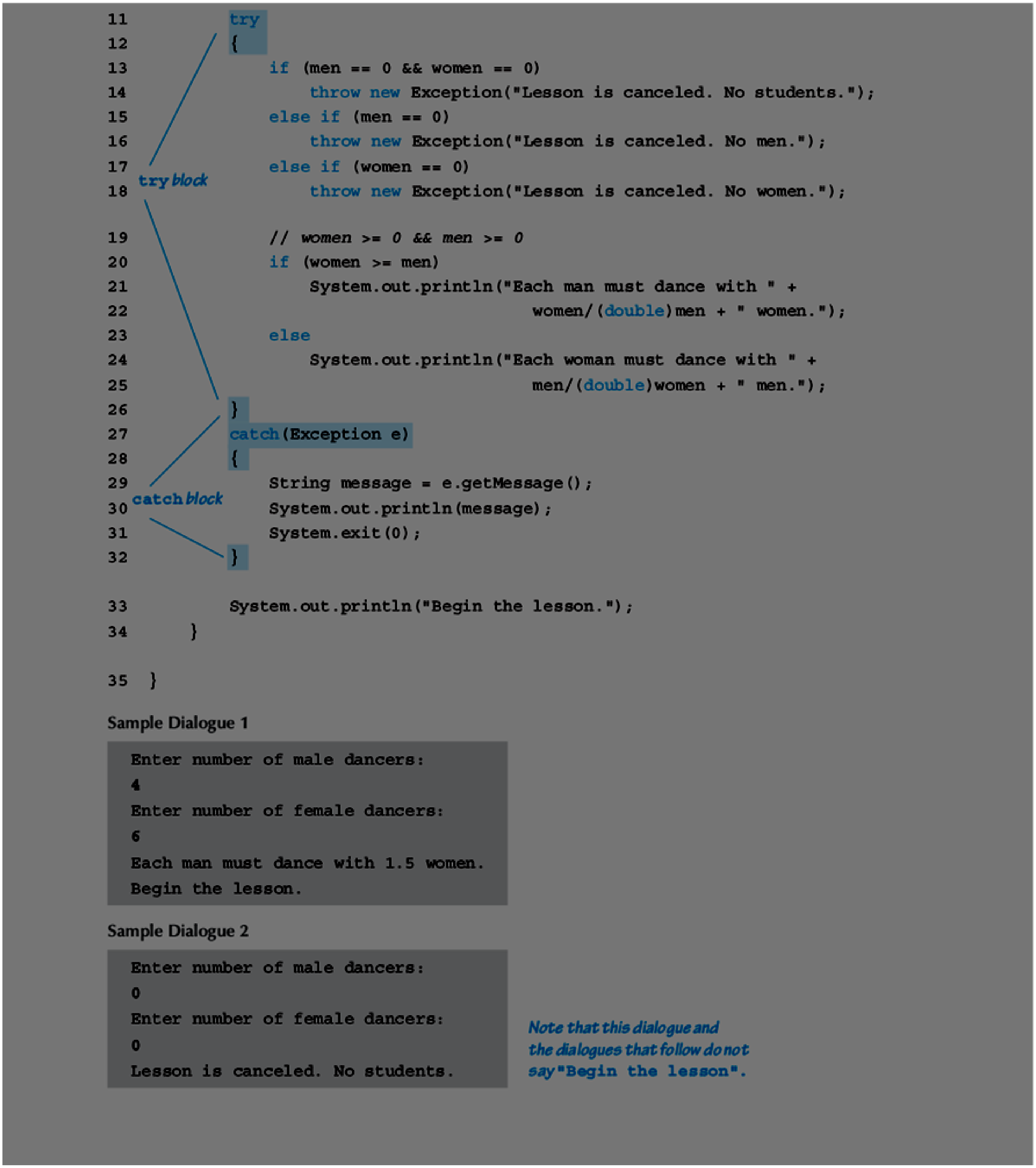
```
import java.util.Scanner;

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();
```

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

(continued)



```
    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.");
  }

}
```

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.
```

Note that this dialogue and the dialogues that follow do not say "Begin the lesson".

Excercise 2:

Q1. What output is produced by the following code?

```
int waitTime = 46;
try
{
   System.out.println("Try block entered.");
   if (waitTime > 30)
      throw new Exception("Over 30.");
   else if (waitTime < 30)
      throw new Exception("Under 30.");
   else
      System.out.println("No exception.");
   System.out.println("Leaving try block.");
}
catch(Exception thrownObject)
{
    System.out.println(thrownObject.getMessage());
}
System.out.println("After catch block");
```

Q2. Suppose that in Q1, the line

```
int waitTime = 46;
is changed to
int waitTime = 12;
```

How would this affect the output?

Q3. What happens when a throw statement is executed? This is a general question. Explain what happens in general, not simply what happens in the code in Exercise 1 or some other sample code.

Q4. Is the following legal?

```
Exception exceptionObject =
new Exception("Oops!");
```

Q5. Is the following legal?

```
Exception exceptionObject =

new Exception("Oops!");

throw exceptionObject;
```

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 the try block, at most one exception will be thrown (since a throw statement ends the execution of the try block), but different types of exception values can be thrown on different occasions when the try block is executed. Each catch block can only catch values of the exception class type given in the catch block heading. However, you can catch exception values of differing types by placing more than one catch block after a try block.

Example

```
import java.util.Scanner;

public class MoreCatchBlocksDemo

{

     public static void main(String[] args){

          Scanner keyboard = new Scanner(System.in);

          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.");
             }
}}
```

Sample Dialogue 1

How many pencils do you have?

5

How many erasers do you have?

2

Each eraser must last through 2.5 pencils

End of program.

Sample Dialogue 2

How many pencils do you have?

−2

Cannot have a negative number of pencils

End of program.

Sample Dialogue 3

How many pencils do you have?

5

How many erasers do you have?

0

Do not make any mistakes.

End of program.

For more details visit following links:

Exceptions in JAVA

Flow Control in Try Catch

Lab Tasks:

Exercise 1

Write a program that calculates the average of N integers. The program should prompt the user to enter the value for N and then afterward must enter all N numbers. If the user enters a no positive value for N, then an exception should be thrown (and caught) with the message " N must be positive." If there is any exception as the user is entering the N numbers, an error message should be displayed, and the user prompted to enter the number again.

Exercise 2

Here is a snippet of code that inputs two integers and divides them:

```
Scanner scan = new Scanner(System.in);

int n1, n2;

double r;

n1 = scan.nextInt();

n2 = scan.nextInt();

r = ( double) n1 / n2;
```

Place this code into a try-catch block with multiple catches so that different error messages are printed if we attempt to divide by zero or if the user enters textual data instead of integers (`java.util.InputMismatchException`). If either of these conditions occurs, then the program should loop back and let the user enter new data.

Exercise 3

Modify the previous exercise so that the snippet of code is placed inside a method. The method should be named ReturnRatio, read the input from the keyboard, and throw different exceptions if there is a division by zero or an input mismatch between text and an integer. Create your own exception class for the case of division by zero. Invoke ReturnRatio from your main method and catch the exceptions in main. The main method should invoke the ReturnRatio method again if any exception occurs.

Exercise 4

Write a program that converts dates from numerical month/day/year format to normal “month day, year” format (for example, 12/25/2000 corresponds to December 25, 2000). You will define three exception classes, one called MonthException , another called DayException , and a third called YearException . If the user entersProgramming Projects anything other than a legal month number (integers from 1 to 12), your program will throw and catch a MonthException and ask the user to reenter the month. Similarly, if the user enters anything other than a valid day number (integers from 1 to either 28 , 29 , 30 , or 31 , depending on the month and year), then your program will throw and catch a DayException and ask the user to reenter the day. If the user enters a year that is not in the range 1000 to 3000 (inclusive), then your program will throw and catch a YearException and ask the user to reenter the year. (There is nothing very special about the numbers 1000 and 3000 other than giving a good range of likely dates.)

Exercise5

Write a program that can serve as a simple calculator. This calculator keeps track of a single number (of type double) that is called result and that starts out as 0.0 . Each cycle allows the user to repeatedly add, subtract, multiply, or divide by a second number. The result of one of these operations becomes the new value of result . The calculation ends when the user enters the letter R for “result” (either in upper- or lowercase). The user is allowed to do another calculation from the beginning as often as desired. The input format is shown in the following sample dialogue. If the user enters any operator symbol other than + , −, * , or / , then an UnknownOperatorException is thrown and the user is asked to reenter that line of input. Defining the class UnknownOperatorException is part of this project.

Sample output:

Calculator is on.

```
result = 0.0

+5

result + 5.0 = 5.0

new result = 5.0

* 2.2

result * 2.2 = 11.0

updated result = 11.0
```

```
% 10

% is an unknown operation.

Reenter, your last line:

* 0.1

result * 0.1 = 1.1

updated result = 1.1

r

Final result = 1.1

Again? (y/n)

yes

result = 0.0

+10

result + 10.0 = 10.0

new result = 10.0

/2

result / 2.0 = 5.0

r

Final result = 5.0

Again? (y/n)

N
```

End of Program

END