CMSC 202

Exceptions

2nd Lecture

Methods may fail for multiple reasons

```
public class BankAccount {
   private int balance = 0, minDeposit = 500;
   public BankAccount( ) {
         balance = 0;
   }

   public int getBalance( ) { return balance; }

   // precondition - amount must be nonnegative an more than min
   // throws an exception if amount is negative
   // postcondition - balance updated
   public int deposit( int amt ) {
         if (amt < 0 )
                  throw new DepositNegativeException( );
         if (amt < minDeposit)
                  throw new DepositTooSmallException( );
         balance += deposit;
   }
```

Multiple `catch` Blocks

- A `try` block can call a method that potentially throws 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

Multiple catch Blocks

```
public class DepositExample2 {
   public static void main( String[ ] args ) {
         BankAccount myAccount = new BankAccount( );
         Scanner input = new Scanner( System.in );
         System.out.print("Enter deposit amount: ");
         int amt = input.nextInt();

         try {
             myAccount.deposit( amt );
             System.out.println( "New Balance = " + myAccount.getBalance());
         }
         catch (DepositNegativeException dne) {
             // code that "handles" a negative deposit
         }
         catch (DepositTooSmallException dts) {
             // code that "handles" a deposit less than the minimum
         }
         System.out.println ("Have a nice day");
}
```

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

Catch the More Specific Exception First

```
public class DepositExample2 {
public static void main( String[ ] args ) {
        BankAccount myAccount = new BankAccount( );
        Scanner input = new Scanner( System.in );
        System.out.print(“Enter deposit amount: “);
        int amt = input.nextInt();

        try {
            myAccount.deposit( amt );
            System.out.println( “New Balance = “ +
                                          myAccount.getBalance());
        }
        catch (Exception e)         // OOOPS!!
        {
            // code to handle an exception
        }
        catch (DepositNegativeException dne) {
            // code that “handles” a negative deposit
        }
        catch (DepositTooSmallException dts) {
            // code that “handles” a deposit less than the minimum
        }
        System.out.println (“Have a nice day”);
}
```

Catch the More Specific Exception First

- Because a `DepositNegativeException` and `DepositTooSmallException` are types of `Exception`, all exceptions will be caught by the first `catch` block before ever reaching the second or third block
  - The catch blocks for `DepositNegativeException` and `DepositTooSmallException` will never be used!
- For the correct ordering, simply put the catch block for Exception last.

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
  ```

```
public int deposit( int amt ) throws DepositNegativeException,
                                     DepositTooSmallException
{
      if (amt < 0 )
              throw new DepositNegativeException( );
      if (amt < minDeposit)
              throw new DepositTooSmallException( );
      balance += deposit;
}
```

The Catch or Declare Rule

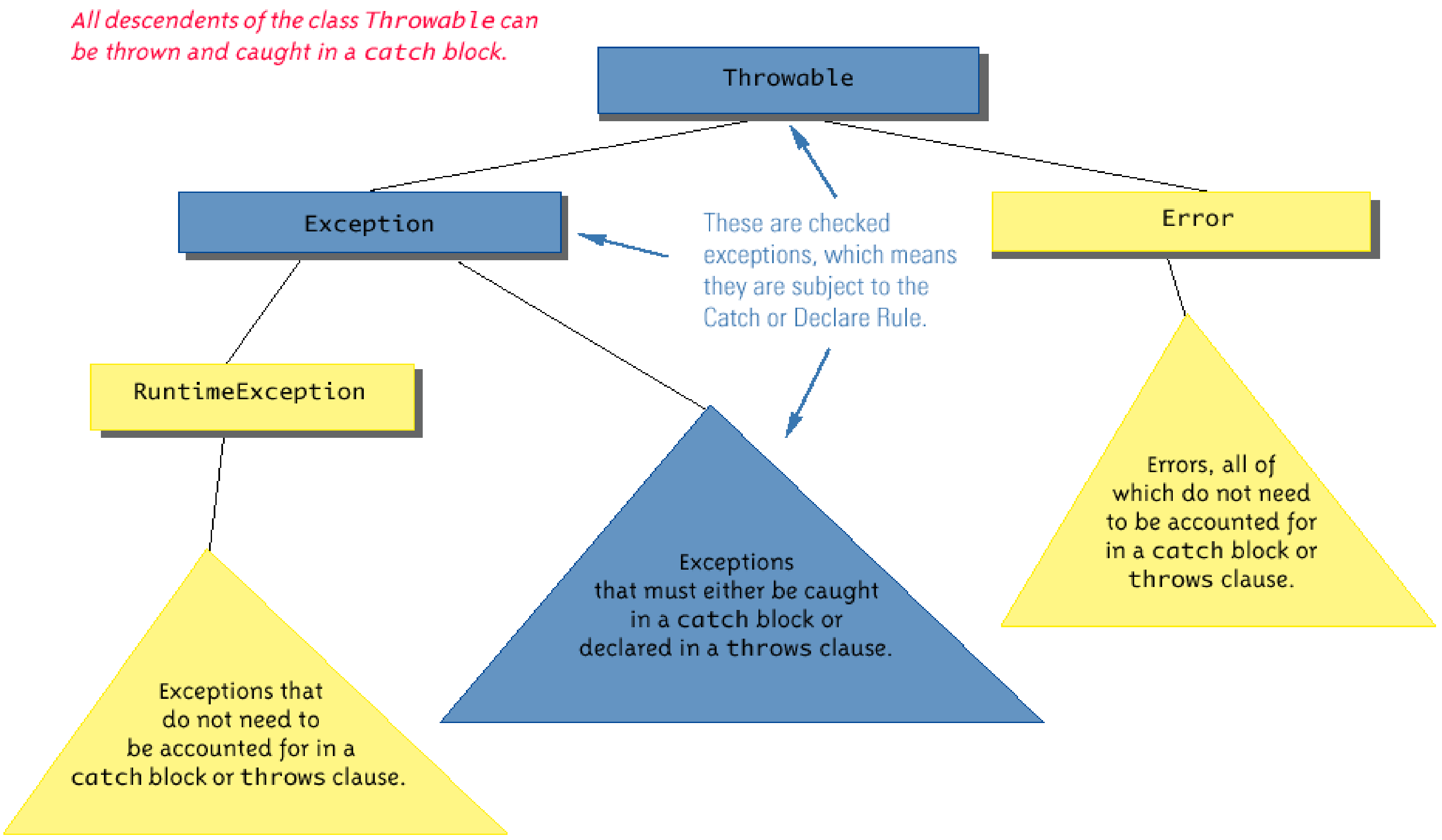
- Most ordinary exceptions that might be thrown within a method must be accounted for in one of two ways:
  1. The code that can throw an exception is placed within a `try` block, and the possible exception is caught in a `catch` block within the same method
  2. The possible exception can be declared at the start of the method definition by placing the exception class name in a `throws` clause

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`** (with the exception of **`RuntimeException`** and its subclasses) are checked exceptions
- All other exceptions are ***unchecked*** exceptions
- The class **`Error`** and all its descendant classes are called *error classes*
  - Error and RuntimeException classes are *not* subject to the Catch or Declare Rule

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 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.
Exceptions that do not need to be accounted for in a catch block or throws clause.

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
- Even if an exception is unchecked, you can still catch if if you want

Runtime Exceptions

- Runtime exceptions are
  - Unchecked
  - Probably a bug in your program
    - Referencing a null pointer
    - Array index out of bounds
  - Thrown automatically by Java

What Happens If an Exception is Never Caught?

- If every method up to and including the main method simply includes a `throws` clause for an exception, that exception may be thrown but never caught
  - In a GUI program (i.e., a program with a windowing interface), nothing happens - but the user may be left in an unexplained situation, and the program may be no longer be reliable
  - In non-GUI programs, this causes the program to terminate with an error message giving the name of the exception class
- Every well-written program should eventually catch every exception by a `catch` block in some method

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

…but most importantly:

  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

When to use a finally block

- The finally block should contain code that you always want to run whether or not an exception occurred.
- Generally the finally block contains code to release resources other than memory
  - Close files
  - Close internet connection
  - Clear the screen

Exception Controlled Loops

- Sometimes it is better to simply loop through an action again when an exception is thrown, as follows. We'll see a real example next.

```
boolean done = false;
while ( ! done )
{
   try
   {
      CodeThatMayThrowAnException
      done = true;
   }
   catch ( SomeExceptionClass e )
   {
      SomeMoreCode
   }
}
```

Exceptions with the `Scanner` Class

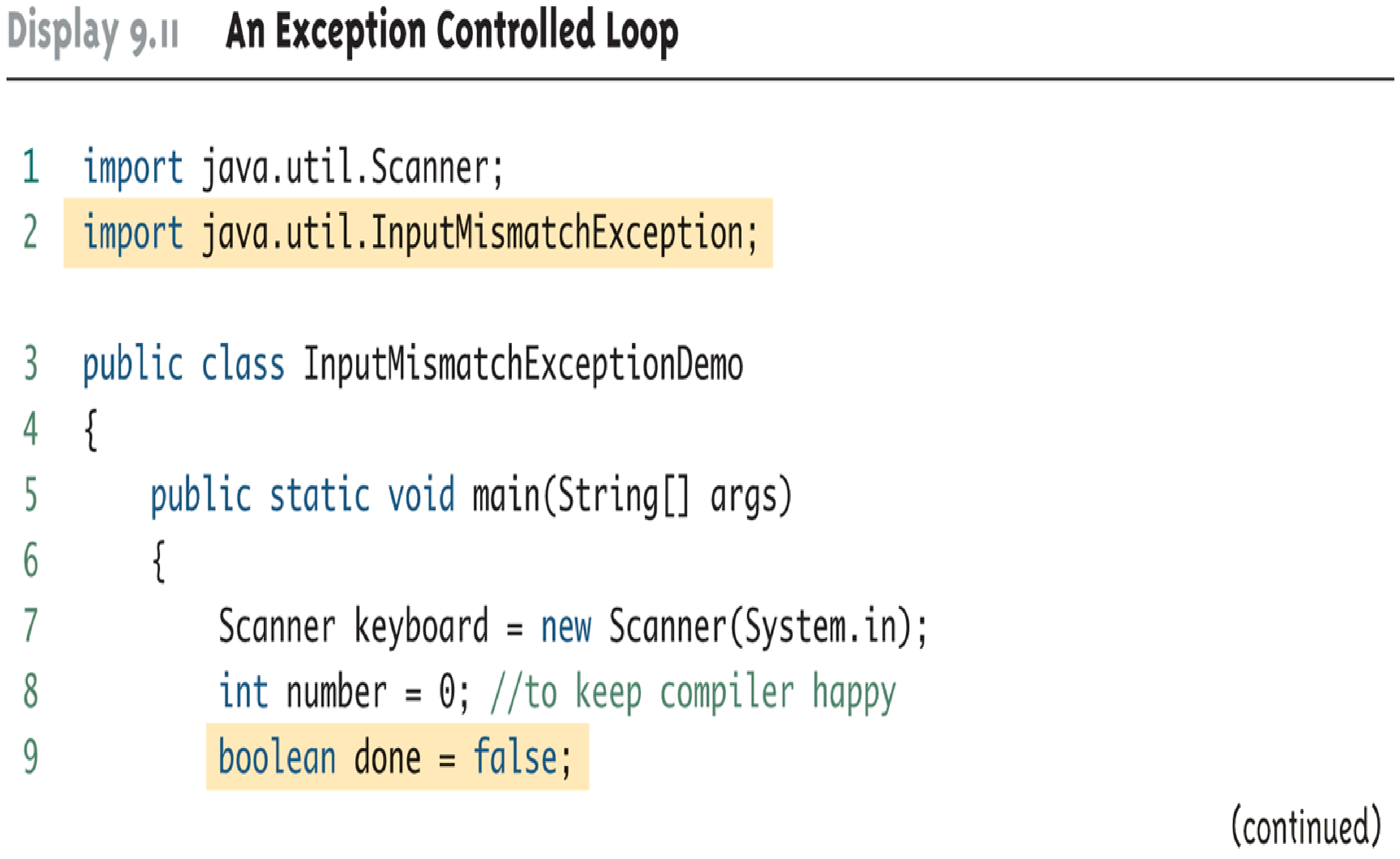
- The `nextInt` method of the `Scanner` class can be used to read `int` values from the keyboard
- However, if a user enters something other than a well-formed `int` value, an `InputMismatchException` will be thrown
  - Unless this exception is caught, the program will end with an error message
  - If 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 (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 (2 of 3)

Display 9.11 An Exception Controlled Loop

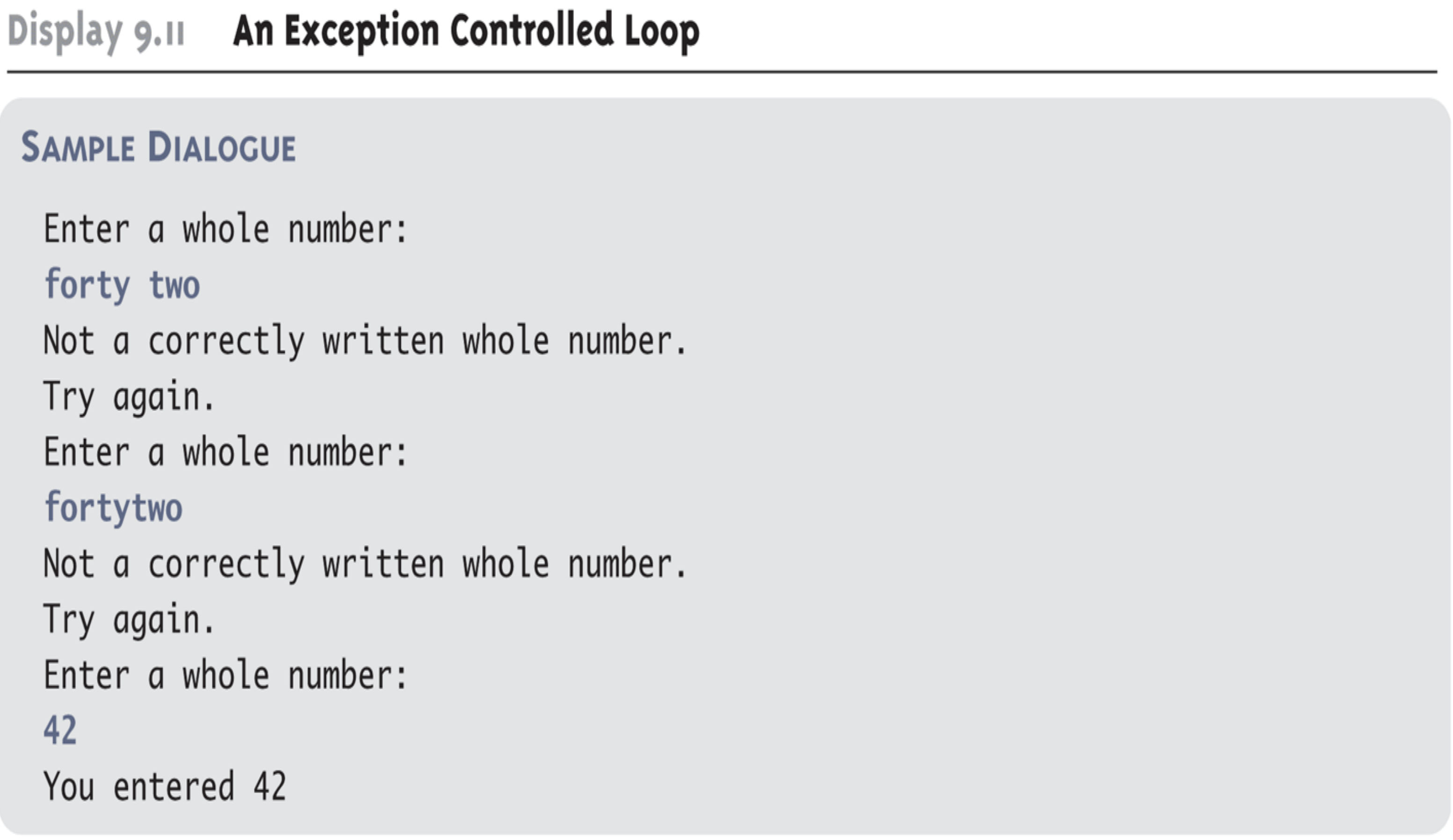
```
        while (! done)
        {
            try
            {
                System.out.println("Enter a whole number:");
                number = keyboard.nextInt();
                done = true;
             }
             catch(InputMismatchException e)
             {
                 keyboard.nextLine();
                 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 (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
```